

SPERRY UNIVAC
Series 1100

FORTRAN (ASCII)


Level 10R1

Programmer Reference

This document contains the latest information available at the time of preparation. Therefore, it may contain descriptions of functions not implemented at manual distribution time. To ensure that you have the latest information regarding levels of implementation and functional availability, please contact your local Sperry Univac representative.

Sperry Univac reserves the right to modify or revise the content of this document. No contractual obligation by Sperry Univac regarding level, scope, or timing of functional implementation is either expressed or implied in this document. It is further understood that in consideration of the receipt or purchase of this document, the recipient or purchaser agrees not to reproduce or copy it by any means whatsoever, nor to permit such action by others, for any purpose without prior written permission from Sperry Univac.

Sperry Univac is a division of the Sperry Corporation.

FASTRAND, SPERRY UNIVAC, UNISCOPE, UNISERVO, UNIVAC, and  are registered trademarks of the Sperry Corporation. ESCORT, MAPPER, PAGEWRITER, PIXIE, and UNIS are additional trademarks of the Sperry Corporation.

THE ASCII FORTRAN LEVEL 10R1 SOFTWARE DESCRIBED IN THIS DOCUMENT IS CONFIDENTIAL INFORMATION AND A PROPRIETARY PRODUCT OF THE SPERRY UNIVAC DIVISION OF SPERRY CORPORATION.

Preface

This manual is for users of ASCII FORTRAN level 10R1. ASCII FORTRAN level 10R1 contains all the features of the FORTRAN standard, X3.9-1978 (also known as FORTRAN 77).

Features of ASCII FORTRAN that are extensions to FORTRAN 77 are shaded. For example, the following nonstandard feature is described in 2.2.3.2:

For syntactic compatibility with other FORTRAN processors, "&" is also allowed as a concatenation operator.

Most of the features that are shaded in this manual also cause a message indicating nonstandard usage to be printed at compilation time if the T option is used (see 10.5.1 and 10.10):

```
CHARACTER*4 A,B*8  
B = A&'END'  
*NON-STD USAGE 3151 at line 2 '&' used as concatenation operator
```

Edit bars, which indicate technical changes, are not used in appendices H, K, and L, since these are new sections.

The following Sperry Univac Series 1100 manuals provide information related to the use of ASCII FORTRAN:

- Executive System, Programmer Reference, UP-4144.2 (applicable version*)
- System Utilities, Programmer Reference, UP-8730 (applicable version*)
- System Relocatable Library and Common Bank (SYSLIB), Programmer Reference, UP-8728 (applicable version*)
- Sort/Merge, Programmer Reference, UP-7621 (applicable version*)
- Collector (MAP Processor), Programmer Reference, UP-8721 (applicable version*)
- Multibanking, Programmer Reference, Preliminary, PUP-8722.P1
- Conversational Time Sharing (CTS), Programmer Reference, UP-7940 (applicable version*)
- Assembly Instruction Mnemonics (AIM), Supplementary Reference, UP-9047 (applicable version*)

* Use the version that reflects the software level used at your site.

Contents

Page Status Summary

Preface

Contents

1. Introduction	1-1
1.1. Reason for FORTRAN	1-1
1.2. Evolution of FORTRAN	1-2
1.3. FORTRAN System	1-2
1.3.1. FORTRAN Processor	1-2
1.3.2. FORTRAN Execution Time System	1-4
1.4. Sample Program	1-4
1.4.1. Explanation of Main Program Coding	1-4
1.4.2. Explanation of Subprogram Coding	1-5
1.4.3. Explanation of System Command Coding	1-6
1.5. Conventions of Notation Used in This Manual	1-6
2. Language Characteristics	2-1
2.1. Character Set	2-1
2.2. Language Elements	2-1
2.2.1. Constants	2-2
2.2.1.1. Integer Constant	2-3
2.2.1.2. Real Constant	2-3
2.2.1.2.1. Single Precision	2-3
2.2.1.2.2. Double Precision	2-4
2.2.1.3. Complex Constants	2-5
2.2.1.4. Logical Constants	2-5
2.2.1.5. Character Constants	2-5
2.2.1.6. Octal and Fielddata Constants	2-6
2.2.2. Symbolic Names	2-6
2.2.2.1. Uniqueness of Symbolic Names	2-7
2.2.2.2. Data Types of Symbolic Names	2-8
2.2.2.2.1. Implied Declaration Via the Name Rule	2-8
2.2.2.2.2. Implied Declaration Using the IMPLICIT Statement	2-9
2.2.2.2.3. Explicit Declaration	2-9
2.2.2.3. Variables	2-9

2.2.2.4. Arrays and Subscripts	2-10
2.2.2.4.1. Array Declaration	2-10
2.2.2.4.2. Value of Dimensions	2-11
2.2.2.4.3. Constant, Adjustable and Assumed-Size Arrays	2-11
2.2.2.4.4. Actual and Dummy Arrays	2-12
2.2.2.4.5. Array Element Reference	2-12
2.2.2.4.6. Location of Elements Within an Array	2-12
2.2.2.5. Character Substrings	2-13
2.2.3. FORTRAN Expressions	2-14
2.2.3.1. Arithmetic Expressions	2-14
2.2.3.1.1. Arithmetic Operators	2-14
2.2.3.1.2. Formation of Arithmetic Expressions	2-15
2.2.3.1.3. Evaluation of Arithmetic Expressions	2-16
2.2.3.1.4. Type Rules for Arithmetic Expressions	2-16
2.2.3.2. Character Expressions	2-17
2.2.3.3. Logical and Relational Expressions	2-18
2.2.3.3.1. Logical Operators	2-18
2.2.3.3.2. Formation of Logical Expressions	2-20
2.2.3.3.3. Evaluation of Logical Expressions	2-20
2.2.3.4. Typeless Expressions	2-21
2.2.3.4.1. Typeless Functions	2-21
2.2.3.4.2. Evaluation of Typeless Expressions	2-22
2.2.3.5. Hierarchy of Operators	2-22
2.2.4. General Statement Form	2-23
2.2.5. Comment Line	2-23
2.2.6. Continuation Line	2-23
2.2.7. Inline Comment	2-24
3. Assignment Statements	3-1
3.1. General	3-1
3.2. Arithmetic Assignment Statement	3-1
3.3. Character Assignment Statement	3-5
3.4. Logical Assignment Statement	3-6
3.5. Statement Label Assignment (ASSIGN Statement)	3-7
4. Control Statements	4-1
4.1. General	4-1
4.2. GO TO Statements	4-2
4.2.1. Unconditional GO TO	4-2
4.2.2. Computed GO TO	4-3
4.2.3. Assigned GO TO	4-5
4.3. IF Statements	4-7
4.3.1. Arithmetic IF	4-7
4.3.2. Logical IF	4-8
4.4. Blocking Statements	4-9
4.4.1. Block IF Statement	4-10
4.4.1.1. IF-Level	4-10
4.4.1.2. IF-Block	4-10
4.4.1.3. Execution of a Block IF Statement	4-10

4.4.2. ELSE IF Statement	4-11
4.4.2.1. ELSE IF-Block	4-11
4.4.2.2. Execution of an ELSE IF Statement	4-11
4.4.3. ELSE Statement	4-11
4.4.3.1. ELSE-Block	4-11
4.4.3.2. Execution of an ELSE Statement	4-12
4.4.4. END IF Statement	4-12
4.4.5. Examples Using the Blocking Statements	4-12
4.5. DO Statement	4-14
4.5.1. Range of a DO-Loop	4-14
4.5.2. Nested DO-Loops	4-14
4.5.3. Active and Inactive DO-Loops	4-15
4.5.4. DO-Loop Execution	4-16
4.5.4.1. Executing a DO Statement	4-16
4.5.4.2. Loop Control Processing	4-17
4.5.4.3. Execution of the Range	4-17
4.5.4.4. Terminal Statement Execution	4-17
4.5.4.5. Incrementation Processing	4-17
4.5.5. Extended Range of a DO-Loop	4-18
4.5.6. Availability of the DO-Variable Value	4-19
4.5.7. DO-Loop Examples	4-19
4.6. CONTINUE Statement	4-21
4.7. PAUSE Statement	4-22
4.8. STOP Statement	4-23
4.9. END Statement	4-24
5. Input/Output Statements	5-1
5.1. General	5-1
5.2. Elements of Input/Output Statements	5-3
5.2.1. File Reference Number Specification	5-3
5.2.2. Record Number Specification	5-4
5.2.3. Input/Output List Specifications	5-4
5.2.4. Format Statement Specification	5-5
5.2.5. Namelist Name Specification	5-6
5.2.6. ERR Clause Specification	5-6
5.2.7. END Clause Specification	5-7
5.2.8. Input/Output Status Clause Specification	5-7
5.3. FORMAT Statement	5-8
5.3.1. Editing Codes	5-9
5.3.2. Editing Code Repetition	5-14
5.3.3. Repetition of Groups of Editing Codes	5-14
5.3.4. Carriage Control	5-14
5.3.5. Complex Variables	5-15
5.3.6. Scale Factor	5-15
5.3.7. Control of Record Handling and List Fulfillment	5-15
5.3.7.1. Multiple Line Formats	5-16
5.3.7.2. End of Input/Output List Test	5-16
5.3.8. Relationships of a Format to an I/O List	5-17
5.3.9. Variable Formats	5-18
5.3.10. Representation of Input/Output Data	5-18

5.4. Namelist	5-19
5.4.1. NAMELIST Statement	5-19
5.4.2. Namelist Input	5-20
5.4.3. Namelist Output	5-22
5.5. List-Directed Input/Output	5-23
5.5.1. List-Directed Input	5-23
5.5.2. List-Directed Output	5-25
5.6. Sequential Access Input/Output Statements	5-26
5.6.1. READ Statements	5-26
5.6.1.1. Formatted READ	5-26
5.6.1.2. Unformatted READ	5-28
5.6.1.3. Namelist READ	5-29
5.6.1.4. List-Directed READ	5-29
5.6.1.5. Reread	5-31
5.6.2. Output Statements	5-32
5.6.2.1. Formatted WRITE	5-32
5.6.2.2. Unformatted WRITE	5-33
5.6.2.3. Namelist WRITE	5-34
5.6.2.4. List-Directed WRITE	5-35
5.6.3. BACKSPACE Statement	5-37
5.6.4. ENDFILE Statement	5-38
5.6.5. REWIND Statement	5-39
5.6.6. Sequential Access DEFINE FILE Statement	5-40
5.7. Direct Access Input/Output Statements	5-44
5.7.1. Direct Access DEFINE FILE Statement	5-44
5.7.2. Direct Access READ Statement	5-46
5.7.3. Direct Access WRITE Statement	5-47
5.7.4. FIND Statement	5-49
5.8. Input/Output Contingencies	5-50
5.8.1. Input/Output Contingency Clauses	5-50
5.8.2. Input/Output Error Messages	5-51
5.9. Internal Files	5-52
5.9.1. Internal File Formatted READ	5-52
5.9.2. DECODE Statement	5-53
5.9.3. Internal File Formatted WRITE	5-55
5.9.4. ENCODE Statement	5-56
5.10. Auxiliary Input/Output Statements	5-58
5.10.1. OPEN	5-58
5.10.2. CLOSE	5-69
5.10.3. INQUIRE	5-71
6. Specification and Data Assignment Statements	6-1
6.1. Overview of Specification Statements	6-1
6.2. DIMENSION Statement	6-2
6.3. Type Statements	6-4
6.3.1. IMPLICIT Statement	6-4
6.3.2. Explicit Type Statements	6-6
6.3.2.1. INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL Type Statements	6-7

6.3.2.2. CHARACTER Type Statement	6-8
6.4. EQUIVALENCE Statement	6-10
6.5. COMMON Statement	6-13
6.6. BANK Statement	6-15
6.7. PARAMETER Statement	6-17
6.8. Initial Value Assignment	6-19
6.8.1. DATA Statement	6-19
6.8.2. Statement Labels	6-20
6.8.3. Octal Constants	6-20
6.8.4. Fielddata Constants	6-20
6.8.5. Other Constants	6-20
6.9. Storage Assignment	6-24
6.9.1. Data Storage Assignment	6-24
6.9.2. Location Counter Usage	6-27
7. Function and Subroutine Procedures	7-1
7.1. Procedures	7-1
7.2. Procedure References	7-2
7.2.1. Function References	7-2
7.2.2. Subroutine References	7-3
7.2.3. EXTERNAL Statement	7-3
7.2.4. INTRINSIC Statement	7-5
7.3. FORTRAN-Supplied Procedures	7-6
7.3.1. Intrinsic Functions	7-6
7.3.2. Pseudo-Functions	7-14
7.3.2.1. BITS and SBITS	7-14
7.3.2.1.1. BITS	7-14
7.3.2.1.2. SBITS	7-15
7.3.2.2. SUBSTR	7-16
7.3.3. Service Subroutines	7-17
7.3.3.1. DUMP	7-17
7.3.3.2. PDUMP	7-18
7.3.3.3. DVCHK	7-19
7.3.3.4. OVERFL	7-20
7.3.3.5. UNDRFL	7-21
7.3.3.6. OVUNFL	7-22
7.3.3.7. UNDSSET	7-23
7.3.3.8. OVFSSET	7-24
7.3.3.9. DIVSET	7-25
7.3.3.10. CMLSET	7-26
7.3.3.11. CHKSV\$ and CHKRS\$	7-28
7.3.3.12. SSWTCH	7-28
7.3.3.13. SLITE	7-29
7.3.3.14. SLITET	7-30
7.3.3.15. EXIT	7-30
7.3.3.16. ERTRAN	7-31
7.3.3.16.1. Input/Output Executive Requests	7-31
7.3.3.16.2. Miscellaneous Executive Requests	7-37
7.3.3.17. NTRAN\$	7-39

7.3.3.17.1. Operations	7-41
7.3.3.17.2. NTRAN\$ Error Messages	7-45
7.3.3.18. CLOSE	7-47
7.3.3.19. FASCFD and FFDASC	7-47
7.3.3.20. MAXAD\$	7-49
7.3.3.21. LOC	7-50
7.3.3.22. MCORF\$ and LCORF\$	7-50
7.3.3.23. F2DYN\$	7-54
7.4. Programmer-Defined Procedures	7-55
7.4.1. Statement Functions	7-55
7.4.1.1. Statement Function Definition Statement	7-56
7.4.1.2. Statement Function References	7-57
7.4.2. Function Subprograms	7-59
7.4.2.1. Structure	7-60
7.4.2.2. FUNCTION Statement	7-60
7.4.3. Subroutines	7-61
7.4.3.1. Structure	7-62
7.4.3.2. SUBROUTINE Statement	7-62
7.5. Function and Subroutine Arguments	7-63
7.6. RETURN Statement	7-65
7.7. ENTRY Statement	7-66
7.8. BLOCK DATA Subprograms	7-68
7.8.1. Structure	7-68
7.8.2. BLOCK DATA Statement	7-68
7.9. PROGRAM Statement	7-69
7.10. Non-FORTRAN Procedures	7-70
7.11. Scope of Names (Local - Global Definitions)	7-70
7.12. SAVE Statement	7-72
8. Program Control Statements	8-1
8.1. General	8-1
8.2. INCLUDE Statement	8-1
8.3. DELETE Statement	8-5
8.4. EDIT Statement	8-6
8.5. COMPILER Statement	8-7
8.5.1. DATA and PARMINIT Options	8-8
8.5.2. BANKED Options	8-9
8.5.3. LINK=IBJ\$ Option	8-10
8.5.4. U1110 = OPT Option	8-10
8.5.5. STD=66 Option	8-11
8.5.6. ARGCHK Options	8-12
8.5.7. PROGRAM=BIG Option	8-12

9. Debug Facility Statements	9-1
9.1. General	9-1
9.2. DEBUG	9-2
9.2.1. UNIT	9-3
9.2.2. SUBCHK	9-3
9.2.3. TRACE	9-4
9.2.4. INIT	9-5
9.2.5. SUBTRACE	9-5
9.3. AT	9-6
9.4. TRACE ON	9-7
9.5. TRACE OFF	9-8
9.6. DISPLAY	9-9
9.7. Debug Facility Example	9-10
10. Writing a FORTRAN Program	10-1
10.1. General	10-1
10.2. FORTRAN Program Organization	10-1
10.2.1. Program Unit	10-1
10.2.2. Types of Program Units	10-1
10.2.3. Program Unit Organization	10-3
10.2.4. Execution Sequence	10-3
10.3. Statement Categories	10-5
10.3.1. Statement Classification	10-5
10.3.2. Ordering of Statements and Lines	10-6
10.4. Source Program Representation and Control	10-7
10.4.1. Source Program Format	10-7
10.4.1.1. Comment Line	10-8
10.4.1.2. Statements	10-8
10.4.1.3. Statement Labels	10-9
10.4.2. Compilation Listing	10-10
10.4.2.1. Listing Options	10-10
10.4.2.2. Composition of a Compilation Listing	10-11
10.4.2.2.1. Identification Line	10-11
10.4.2.2.2. Source Code Listing	10-11
10.4.2.2.3. Cross Reference Listing	10-17
10.4.2.2.4. Object Code Listing	10-18
10.4.2.2.5. Storage Assignment Map	10-19
10.4.2.2.6. Common Block Listing	10-19
10.4.2.2.7. Entry Point Listing	10-20
10.4.2.2.8. External References	10-20
10.4.2.2.9. Termination Message	10-20
10.5. Calling the ASCII FORTRAN Processor	10-21
10.5.1. Processor Call Options	10-21
10.5.2. Execution of the Object Program	10-24
10.5.2.1. Execution Using Checkout	10-24
10.5.2.2. Collection and Execution	10-24

10.6. FORTRAN Checkout Mode	10-25
10.6.1. Calling Checkout Mode	10-25
10.6.2. Interactive Debug Mode in the Checkout Compiler	10-26
10.6.2.1. Entering Interactive Debug Mode	10-26
10.6.2.2. Soliciting Input	10-27
10.6.3. Debug Commands	10-27
10.6.3.1. BREAK	10-29
10.6.3.2. CALL	10-30
10.6.3.3. CLEAR	10-32
10.6.3.4. DUMP	10-33
10.6.3.5. EXIT	10-34
10.6.3.6. GO	10-35
10.6.3.7. HELP	10-36
10.6.3.8. LINE	10-37
10.6.3.9. LIST	10-37
10.6.3.10. PROG	10-38
10.6.3.11. RESTORE	10-39
10.6.3.12. SAVE	10-41
10.6.3.13. SET	10-42
10.6.3.14. SETBP	10-43
10.6.3.15. SNAP	10-44
10.6.3.16. STEP	10-45
10.6.3.17. TRACE	10-45
10.6.3.18. WALKBACK	10-46
10.6.4. Contingencies in Checkout Mode	10-48
10.6.5. Checkout Mode Restrictions	10-49
10.6.6. Restart Processor (FTNR)	10-49
10.7. Walkback and the Interactive FTNPMD	10-50
10.7.1. Introduction	10-50
10.7.2. Diagnostic Tables Generated by ASCII FORTRAN	10-50
10.7.3. Initiating FTNWB and FTNPMD	10-51
10.7.4. Walkback (FTNWB)	10-52
10.7.4.1. Description of the Walkback Process	10-52
10.7.4.1.1. Errors Detected by the Math Library (CML)	10-52
10.7.4.1.2. Errors Detected by the I/O Library	10-53
10.7.4.1.3. Errors Detected in the User Program	10-55
10.7.4.1.4. FTNWB Routine Call	10-56
10.7.4.2. Walkback Messages	10-57
10.7.4.3. Walkback Procedures for MASM Subprograms	10-58
10.7.4.3.1. F\$EP	10-58
10.7.4.3.2. F\$INFO	10-58
10.7.4.3.3. Description	10-59
10.7.5. Interactive Postmortem Dump (FTNPMD)	10-61
10.7.5.1. Soliciting Input	10-61
10.7.5.2. PMD Mode Commands	10-61
10.7.5.2.1. DUMP	10-61
10.7.5.2.2. EXIT	10-64
10.7.5.3. FTNPMD Diagnostics	10-65
10.8. Compiler Optimization	10-67
10.8.1. Local Optimization	10-67
10.8.2. Global Optimization	10-68
10.8.3. Optimization Pitfalls	10-68
10.9. Hints for Efficient Programming	10-69

10.10. Diagnostic System	10-70
Appendix A. Differences Between SPERRY UNIVAC FORTRAN Processors	A-1
A.1. General	A-1
A.2. Extensions to SPERRY UNIVAC FORTRAN V	A-1
A.3. Exceptions to SPERRY UNIVAC FORTRAN V	A-3
A.4. Differences in Syntax	A-8
Appendix B. ASCII Symbols and Codes	B-1
Appendix C. Programmer Check List	C-1
C.1. General	C-1
C.2. Language Errors	C-1
C.3. Techniques	C-3
Appendix D. Diagnostic Messages	D-1
Appendix E. Conversion Table	E-1
Appendix F. Tables of FORTRAN Statements	F-1
Appendix G. ASCII FORTRAN Input/Output Guide	G-1
G.1. General	G-1
G.2. System Data Format (SDF) File	G-2
G.2.1. SDF File Description	G-2
G.2.1.1. SDF Labels	G-3
G.2.1.2. SDF Data Records/Record Segments	G-5
G.2.1.3. SDF Block Size	G-6
G.2.1.4. SDF End-of-File Record	G-6
G.2.1.5. SDF File Layout	G-7
G.2.2. SDF File Processing	G-7
G.2.2.1. Sequential Access	G-7
G.2.2.2. Direct Access	G-8
G.2.3. SDF Files Not Written by Processor Common I/O	G-9
G.3. ANSI Magnetic Tape Interchange Format	G-10
G.3.1. ANSI File Description	G-10
G.3.2. ANSI File Processing	G-13
G.3.3. ANSI Interchange Tapes from Other Systems	G-13
G.4. ASCII Symbiont Files	G-14
G.5. Unit Reference Number and File Assignment	G-15
G.6. File Reference Table Element - F2FRT	G-16

G.7. Free Core Area Element – F2FCA	G-18
G.8. Storage Control Table Element – F2SCT	G-19
G.9. Input/Output Errors	G-20
G.9.1. ERR Clause Specified	G-20
G.9.2. ERR Clause Not Specified	G-21
G.9.3. Error Clause Listing	G-21
G.10. FORTRAN DEFINE FILE Block Usage	G-32
G.11. Storage-Allocation Packet (Element M\$PKT\$)	G-34
Appendix H. Large Programs and the Multibanking Features of ASCII FORTRAN	H-1
H.1. Large Programs	H-1
H.2. Banking	H-1
H.2.1. General Banking Example (Dual-PSR System)	H-2
H.2.1.1. Collection of the General Banking Example	H-4
H.2.1.2. Analysis of the Collector Symbolic	H-6
H.2.1.3. Large Banks	H-7
H.2.1.4. Variations on the Dual-PSR Structure	H-8
H.2.2. Banking for Single-PSR 1100 Systems	H-10
H.2.3. Banking, Efficiency, and Source Program Directives	H-12
H.2.3.1. I-Bank Linkages	H-12
H.2.3.2. D-Bank Linkages	H-12
H.2.3.3. Multiple I-Banks Only	H-13
H.2.3.4. Multiple Paged Data Banks	H-13
H.2.3.4.1. The BANK Statement	H-13
H.2.3.4.2. Optimization and Program Organization	H-14
H.2.4. Banking Summary	H-15
Appendix I. Error Diagnostics in Checkout Mode	I-1
Appendix J. Comparison of ASCII FORTRAN Level 8R1 to Level 9R1 and Higher	J-1
J.1. General	J-1
J.2. FORTRAN Terms and Concepts	J-1
J.3. Characters, Lines, and Execution Sequence	J-2
J.4. Data Types and Constants	J-2
J.5. Arrays and Substrings	J-2
J.6. Expressions	J-3
J.7. Executable and Nonexecutable Statement Classification	J-3
J.8. Specification Statements	J-3
J.9. DATA Statement	J-6

J.10. Assignment Statements	J-6
J.11. Control Statements	J-6
J.12. Input/Output Statements	J-7
J.13. Format Specification	J-7
J.14. Main Program	J-8
J.15. Functions and Subroutines	J-8
J.16. Block Data Subprogram	J-9
Appendix K. Interlanguage Communication	K-1
K.1. ASCII FORTRAN (FTN) to SPERRY UNIVAC FORTRAN V	K-1
K.2. ASCII FORTRAN to PL/I	K-2
K.2.1. Restrictions and Considerations	K-2
K.2.2. PL/I Argument Counterparts	K-3
K.3. ASCII FORTRAN to ASCII COBOL (ACOB)	K-4
K.3.1. ASCII COBOL Argument Counterparts	K-5
K.4. ASCII FORTRAN and MASM Interfaces	K-6
K.4.1. Arguments	K-6
K.4.2. ASCII FORTRAN Register Usage	K-8
K.4.3. Initializing the ASCII FORTRAN Environment	K-8
K.4.4. Terminating the ASCII FORTRAN Environment	K-9
K.4.5. Calling an ASCII FORTRAN Subprogram	K-9
K.4.6. ASCII FORTRAN Function References	K-10
K.4.7. Example	K-11
Appendix L. ASCII FORTRAN Sort/Merge Interface	L-1
L.1. General	L-1
L.2. Sort/Merge Features Available Through ASCII FORTRAN	L-1
L.3. Restrictions with Sort/Merge Interface	L-1
L.3.1. Banked Arguments Not Allowed	L-1
L.3.2. Sort/Merge Interface Contains Only Formatted I/O	L-1
L.3.3. Use of ASCII FORTRAN Free Core Area Element (F2FCA)	L-2
L.4. The CALL Statement to FSORT	L-2
L.4.1. The CALL Statement for a Sort	L-2
L.4.2. Examples of Sort with Logical Unit Numbers	L-9
L.4.3. Examples of Sort with User Subroutines	L-10
L.5. The CALL Statement to FMERGE	L-12
L.5.1. The CALL Statement for a Merge	L-12
L.5.2. Examples of CALL Statements to Merge	L-16
L.6. The CALL Statement to FSCOPY	L-17
L.6.1. The CALL Statement to Copy an External Sort Parameter Table	L-17
L.6.2. Record Size When FSCOPY Is Used	L-18
L.6.3. An Example of CALL Statement to FSCOPY	L-18

L.7. The CALL Statement to FSSEQ	L-19
L.7.1. The CALL Statement to Provide a User-Specified Collating Sequence	L-19
L.7.2. An Example of the CALL Statement to FSSEQ	L-19
L.8. User-Specified Subroutines	L-21
L.8.1. User-Specified Input Subroutine	L-21
L.8.1.1. An Example of a User-Specified Input Subroutine	L-21
L.8.1.2. The CALL Statement to FSGIVE	L-22
L.8.1.3. An Example of User-Specified Input Subroutine with FSGIVE	L-22
L.8.2. A User Comparison Routine	L-23
L.8.2.1. An Example of a User Comparison Subroutine	L-23
L.8.2.2. An Example of a Runstream with a Comparison Subroutine	L-24
L.8.3. User Data Reduction Subroutine	L-25
L.8.3.1. A Simple Example of a Data Reduction Subroutine	L-25
L.8.3.2. An Example of a Runstream with a Data Reduction Subroutine	L-26
L.8.4. User-Specified Output Subroutine	L-27
L.8.4.1. A Simple Example of a User-Specified Output Subroutine	L-27
L.8.4.2. The CALL Statement to FSTAKE	L-27
L.8.4.3. An Example of FSTAKE in an Output Subroutine	L-27
L.9. Optimizing Sorts	L-28
L.9.1. The Bias of the Input Data	L-29
L.9.2. The Size of the Main Storage Scratch Area	L-29
L.9.2.1. The Use of R\$CORE	L-29
L.9.2.2. The Use of the CORE Clause in the Information String	L-29
L.9.3. The Scratch Files Used and Checksum	L-30
L.9.3.1. Scratch Files Named in the Information String	L-30
L.9.3.2. Checksum and the Sort	L-30
L.10. Sorting Very Large Amounts of Data	L-31
L.10.1. An Example of a Large Single Cycle Sort	L-31
L.10.2. An Example of a Multiple Cycle Sort	L-32
L.11. Error Messages from a Sort or a Merge	L-33

Index

User Comment Sheet

Figures

Figure 1-1. Compilation Process	1-3
Figure 1-2. Sample Program	1-5
Figure 8-1. Sample PROC	8-3
Figure 10-1. Program Units Within a FORTRAN Program	10-2
Figure 10-2. Sample Control Paths During Execution	10-4
Figure 10-3. Order of Statements and Lines	10-7
Figure 10-4. L Option Listing	10-12

Tables

Table 2-1. ASCII FORTRAN Character Set	2-2
Table 2-2. Composition of an Arithmetic Term	2-15
Table 2-3. Length and Type of Result for Arithmetic Expressions	2-17
Table 2-4. Pure Logical Operators	2-19

Table 2-5. Relational Operators	2-19
Table 2-6. Composition of Logical Term	2-20
Table 3-1. Conversions for Arithmetic Assignment	3-3
Table 3-2. Conversions for Character Assignment	3-5
Table 6-1. Valid Data Type and Length	6-4
Table 6-2. Storage Units Required for Data Storage	6-11
Table 6-3. Data Initializations	6-21
Table 6-4. Storage Alignment and Requirement	6-25
Table 6-5. Location Counter Usage	6-27
Table 7-1. Typeless Intrinsic Functions	7-6
Table 7-2. Intrinsic Functions	7-8
Table 8-1. Use of the STD=66 COMPILER Statement Option	8-12
Table 10-1. Processor Option Letters	10-22
Table B-1. ASCII Control Characters	B-1
Table B-2. Graphic ASCII Characters	B-2
Table E-1. Conversion Methods for Arithmetic Data	E-2
Table F-1. Nonexecutable Statements	F-1
Table F-2. Executable Statements	F-2
Table I-1. Messages Occurring During Program Load	I-1
Table I-2. Messages Generated by Interactive Debugging	I-2

1. Introduction

1.1. Reason for FORTRAN

FORTRAN (from FORMula TRANslator) is a programming language designed for extensive use in mathematical, scientific, and technological areas. The advantages of FORTRAN are minimum programming time and cost, and maximum interchangeability of FORTRAN programs on different FORTRAN processors.

FORTRAN statements resemble English statements and the equations of elementary algebra. Therefore, FORTRAN statements are self-documenting since the intended operation is apparent from the statement itself. For example, to find the average of two numbers, the programmer can write a statement such as:

$$\text{AVRGE} = (\text{A} + \text{B}) / 2.0$$

Since the FORTRAN programmer uses a programming language that resembles the language ordinarily used for the solution of problems, relatively little time is required to learn the language. As a result, programming effort can be devoted to the logic of the problem without being troubled by the intricacies of computer operation. The self-documenting feature of FORTRAN reduces debugging time and enables other programmers to grasp readily the logic of a program so that it can be modified or adapted to other purposes with minimal effort.

SPERRY UNIVAC Series 1100 FORTRAN (ASCII) handles the full American Standard Code for Information Interchange (ASCII) character set. Throughout this manual, the language will be referred to as ASCII FORTRAN.

A FORTRAN program written for a particular FORTRAN processor can be accepted by many different FORTRAN processors with a minimum of change. ASCII FORTRAN has been written in accordance with the specifications of the American National Standards Institute, Inc. (ANSI), in ANSI X3.9-1978 (also known as FORTRAN 77).

The ASCII FORTRAN language is a superset of the standard FORTRAN language defined by ANSI X3.9-1978. The ASCII FORTRAN language is highly compatible with SPERRY UNIVAC Series 1100 FORTRAN V. (See Appendix A for a comparison of these two languages.)

1.2. Evolution of FORTRAN

The fundamental unit of information handled by a data processing system is the bit (from binary digit). Basic machine language words are composed of a sequence of bits. A word may be interpreted by the computer as an instruction or as data. The meanings of the bits in instruction words (machine language) are highly specific to each computer. Each bit has two mutually exclusive states, represented by 0 and 1.

The next logical step from machine language is what is commonly known as assembly language. Assembly language requires a language translation program (an assembler). Using assembly language, the programmer is permitted to use symbolic references to storage and specific mnemonic codes instead of numeric codes to designate the operation to be performed.

FORTRAN is one of many higher level languages that has evolved from assembly language. It is considered a higher level language because the translation of a single FORTRAN statement may result in many machine language instructions. This conversion is performed by a program called the FORTRAN processor. The design of a FORTRAN processor is definitely machine-oriented and is not part of the FORTRAN language. Offsetting this processor complexity is the decreased programming time required for learning, writing, debugging, and maintenance.

1.3. FORTRAN System

The FORTRAN system is composed of three main entities: the program, the language processor, and the execution-time system.

Use of the word program is meant to include the source program and the object program. The source program is what is written by a programmer to solve a particular problem. For the computer to understand the source program, it must be translated by the FORTRAN processor into machine language. This output of the FORTRAN processor is called an object program.

1.3.1. FORTRAN Processor

The main function of the ASCII FORTRAN processor is to prepare a machine language object program from the source program code. Generally the processor is referred to as a compiler.

The source program is composed of the FORTRAN statements, written by the programmer, which represent logical steps for solving a particular problem. The organization of a source program is discussed in detail in Section 10.

The compiler makes use of the overall logic structure of the program and generates machine instructions for each FORTRAN statement contained in the programmer-written source program. To produce a machine-acceptable object program, it also assigns storage locations for variables and constants, and creates references to external programs and variables, when required.

The ASCII FORTRAN compiler is a multiphase, modular compiler. The phases have been separated, on the basis of general operations, into a set of reentrant instruction banks and a single, nonreentrant, data bank. The separation into banks means that only those portions of the compiler necessary to process the user's program are loaded. Because the instruction banks are reentrant, any number of users may simultaneously be executing instructions from a single bank of the compiler. Thus, the compiler makes efficient use of resources, especially in an active multiprogramming environment.

The compiler reads the source program from a series of statement lines present in a runstream or saved in a program file. Comment lines may be used freely without affecting the compilation.

The main output of a compilation consists of a relocatable binary program. This object program contains, in binary coded form, the names of all common blocks, external functions, and external subroutines referenced in the source program, and the machine code for the source program. ASCII FORTRAN generates reentrant I-bank code (that is, the code does not modify itself).

Optionally, the compiler may be directed to generate code in main storage and immediately execute it. This mode of operation is the FORTRAN checkout mode, during which additional diagnostic facilities are provided. Throughput is increased during this mode when the programs being processed are to be executed only once without modification or when the execution time is relatively short. No relocatable binary program is produced in this case (see 10.5 and 10.6).

A separate restart processor, FTNR, is available in conjunction with FORTRAN checkout mode. FTNR allows the user to reenter execution of a previously saved program (see checkout debug SAVE command, 10.6.3.12), without recompiling the source program. For further information, see 10.6.6.

Much of the compiler's effort is devoted to the detection of source language errors. If any errors are detected, or remarks are to be made, the compiler prints diagnostic comments adjacent to the FORTRAN statements containing these errors. The compilation listing produced is controlled by several options which determine the amount of information to be printed. The compilation listing is described in 10.4.2.

Using processor options, the compiler may also be directed to devote additional time to optimizing the FORTRAN statements, before generating the relocatable binary output. This is done at the expense of compilation speed, but the resulting output will usually execute significantly faster than without this additional optimization. The various types of optimization are explained in 10.8.

The compilation process (if not in checkout mode) is illustrated in Figure 1-1.

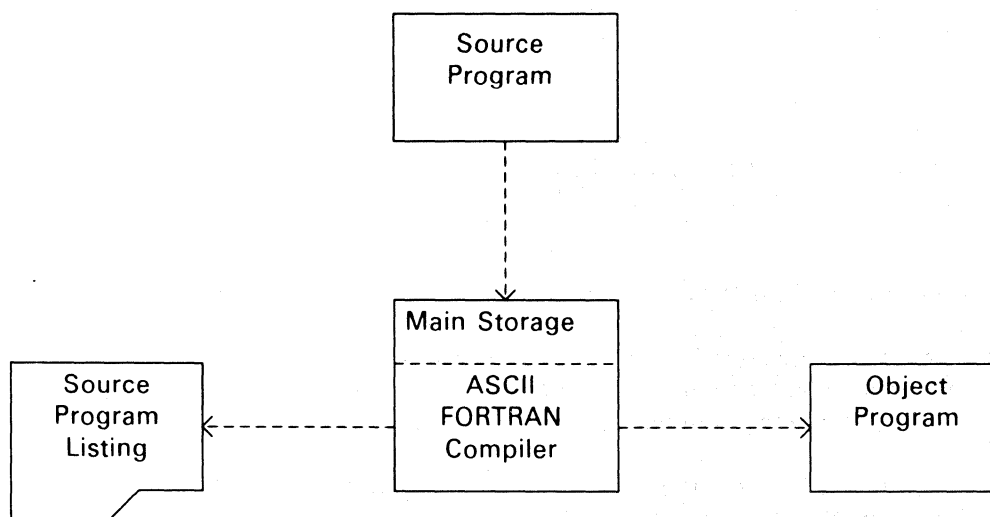


Figure 1-1. Compilation Process

1.3.2. FORTRAN Execution Time System

The FORTRAN execution time system consists of the system software, the compiler and library elements, the program, the computer itself, and any peripheral devices attached to the computer. The system software is referred to as the operating system. It may contain programs which control scheduling, file management, input/output, compilation, debugging, storage assignment, linking, loading, assembly, and other necessary functions.

1.4. Sample Program

The following simple executable program (Figure 1-2) shows FORTRAN programming and terminology. The concepts and terminology used in the description of the program are described in detail in other sections of the manual. How to organize a FORTRAN program is discussed in 10.2. This sample program consists of two program units: the main program, and the external function AMEAN.

This program calculates the average of a series of numbers. The subprogram is generalized in order to calculate the average of a variable number of values. The subprogram results are printed in the main program together with explanatory text.

This is not the only program that could have been written for the problem, nor is it the shortest in terms of lines required. It does introduce the general framework of the FORTRAN system and some of the nomenclature. The number in parentheses preceding each statement is for reference purposes; it is not part of the program. The delta (Δ) symbol indicates a significant blank.

1.4.1. Explanation of Main Program Coding

Line 4 is a comment line indicated by the character C in column 1. This comment consists of all blank characters producing a blank line (except that the C is printed). Comment lines are not required, but are included to aid the reader in understanding the program.

Line 5 of the program is also a comment line. This line (including the C) is printed when the program is compiled, but does not affect execution of the program. Comment lines provide documentation for the programmer.

Line 8 indicates that the symbolic name AMEAN is the name of an external function.

Line 9 causes the ASCII FORTRAN compiler to set aside five locations for array A. A is single precision real.

Line 10 sets initial values for A (symbolic name for the value of which the mean is to be computed). The DATA statement does this at compilation time rather than execution time, in order to reduce execution time for the program.

Line 11 contains a FORMAT statement which specifies that the characters " AVERAGE IS:" are to be printed, followed by five print positions for the value returned from the function AMEAN. This value is to be printed with two digits to the right of the decimal point.

Line 12 assigns the average of array A to variable AVE. The average is obtained from function AMEAN.

Line 14 is a PRINT statement which specifies that the value stored in AVE is to be printed according to the FORMAT statement prefixed by the identifying number 1. Sample output would be:

```
AVERAGEΔIS:Δ3.00
```

Line 16 indicates the end of the main program.

```
(1) @RUN SAMPL,999999,SMW
(2) @ASG,PC MY*FILE1
(3) @FTN,SI MY*FILE1.MAIN,TPF$.MAIN
(4) C
(5) C COMPUTE AVG OF NUMBERS TO BE INPUT FROM DATA
(6) C
(7) C INITIALIZE
(8)     EXTERNAL AMEAN
(9)     REAL A(5)
(10)    DATA A/1.,2.,3.,4.,5./
(11) 1   FORMAT('ΔAVERAGEΔIS:',F5.2)
(12)    AVE = AMEAN (A,5)
(13) C PRINT AVERAGE AND TEXT
(14)    PRINT 1,AVE
(15) C INDICATE END OF MAIN PROGRAM
(16)    END
(17) C FOLLOWING FUNCTION IS CALLED BY MAIN PROGRAM
(18)    FUNCTION AMEAN(DATA,N)
(19)    DIMENSION DATA (N)
(20)    SUM = 0
(21)    DO 1 I = 1,N
(22) 1   SUM = SUM+DATA(I)
(23)    AMEAN = SUM/N
(24) C INDICATE END OF FUNCTION
(25)    RETURN
(26)    END
(27) @XQT
(28) @FIN
```

Figure 1-2. Sample Program

1.4.2. Explanation of Subprogram Coding

Line 18 identifies an external function subprogram named AMEAN with inputs of DATA and N. This procedure computes the average of N numbers and then returns it to the referencing program.

Line 19 is a specification statement. It declares the input variable DATA to be an array. The value N indicates the array is variable in length, and that the actual length is determined by the main program.

Line 20 is an arithmetic assignment statement initializing the variable SUM to zero.

Line 21 controls the repeated execution of line 22. In this sample, the statement labeled 1 (line 22) is repeated five times since N has been given the value five by the calling program. The variable I

initially has the value of one. This is incremented by one each time the set consisting of lines 21 and 22 is repeated.

Line 22 is an arithmetic assignment statement that updates the running total. It obtains the current value of SUM, adds to it the I^{th} number in array DATA, and assigns this sum as the new value for SUM. Line 22 is repeated five times in this sample. Each time the statement is executed, a new value is obtained from the input array DATA, starting at DATA(1) and ending with DATA(5).

Line 23 computes the average as the sum divided by the number of elements, and assigns the result as the value of function AMEAN.

Line 25 returns control from function AMEAN to the statement in the main program which referred to or called this function. In this sample, the return is to line 12.

Line 26 indicates the end of the function subprogram.

Note that every statement (columns 7 through 72), except the assignment statements, starts with a keyword. (Comment lines are not considered statements.) The keyword is an English word that describes the purpose of the statement. Every statement in ASCII FORTRAN, except statement functions and certain assignment statements, begins with a keyword. Keywords are not reserved words in ASCII FORTRAN. They may be used anywhere in the program as symbolic names.

1.4.3. Explanation of System Command Coding

Line 1 of Figure 1-2 is a RUN statement. It must be the first Executive control command in a run. It identifies the run to the system and supplies accounting information.

Line 2 is an Executive control statement. It is used to assign an external file to the run under the given name. It can state input/output requirements for this file. (See EXEC Programmer Reference, UP-4144.2 (see Preface).)

Line 3 is the ASCII FORTRAN language processor system command for the compilation of the main program and function AMEAN. It contains information on the location of source input, the type of listing desired and the placement of relocatable binary code after compilation. (See 10.5.)

Line 27 is an execute command. It causes the program to be collected (if not previously specified in a @MAP control statement) with the ASCII FORTRAN relocatable library elements in SYS\$*RLIB\$. If these elements are not in SYS\$*RLIB\$, a specific collection (@MAP statement) must be done before the program can be executed (@XQT statement).

Line 28 signals the termination of the run.

1.5. Conventions of Notation Used in This Manual

Throughout this manual instructive examples are furnished to illustrate syntax or other material covered in each section. The comments associated with these examples are shown as FORTRAN comment statements, with C or * in the first position of each line.

Where the format of a statement or clause is discussed, optional entities or clauses are indicated by their enclosure in the bracket pair ([]). Allowable repetition of entities or clauses is indicated by the ellipsis (...). Items to be filled in with program information appear in italics. Variables in example comments are also italicized for clarity in discussion.

2. Language Characteristics

2.1. Character Set

The ASCII FORTRAN programming character set, as distinguished from the literal data set, consists of 26 letters (uppercase and lowercase), 10 digits, and 16 special characters from the ASCII character set. (Note that use of lowercase letters and other nonstandard characters will not result in a *NON-STD USAGE message. See 10.10).

The compiler reads all input in the 9-bit character form of the American Standard Code for Information Interchange (ASCII). All character literal data is retained in this form by the compiler. The 9-bit representation of ASCII consists of one zero bit followed by the 8-bit ASCII code.

The literal string 'abcd' is distinct from 'ABCD' in internal representation. In other instances the compiler will consider lowercase alphabetic characters to be identical with uppercase alphabetic characters. Thus, the symbolic name xxx is identical to the symbolic name XXX. Source output is in the ASCII character set unless explicitly directed to be Fieldata (see 10.5). If source input is in Fieldata, however, source output will remain in Fieldata.

The ASCII FORTRAN character set is summarized in Table 2-1. The ASCII code set is given in Appendix B.

2.2. Language Elements

Combinations of FORTRAN character set members form FORTRAN language elements. These, in turn, are used to form statements. An appropriate set of statements forms a program unit (see 10.2.3).

The main language elements are comments, constants, variables, arrays, and operators.

Table 2-1. ASCII FORTRAN Character Set

Character Group	Members
Uppercase and Lowercase Alphabets	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z
Digits	0 1 2 3 4 5 6 7 8 9
Special Characters	blank (represented by "Δ" in this manual)
	= (equals)
	+ (plus)
	- (minus)
	* (asterisk)
	/ (slash)
	((left parenthesis)
) (right parenthesis)
	, (comma)
	. (decimal point)
	: (colon)
	' (apostrophe)
	< (less than)
	> (greater than)
	\$ (currency symbol)
	& (ampersand)

2.2.1. Constants

A constant is an item whose value is determined from its name and initial value. The value of a constant cannot be changed.

The constants that may be used in FORTRAN fall into the following four categories:

- Numeric
 - Integer
 - Real
 - Double precision real
 - Complex
 - Double precision complex
- Logical
- Literal (Hollerith or character)
- Octal

Sometimes, constants are referred to as scalars. A scalar is a single item. Constants and single variables may be referred to as scalars. Arrays are not scalars.

2.2.1.1. Integer Constant

The absolute value of an integer constant must be less than or equal to:

$$2^{35} - 1 = 34,359,738,367$$

An integer is represented internally as a fixed point number, occupying one word of storage. It must not contain a decimal or comma. It may assume a positive, negative, or zero value. Therefore, it may be prefixed with a plus or minus sign. The following are valid integer constants:

```
0
+753
-999999
2501
```

2.2.1.2. Real Constant

2.2.1.2.1. Single Precision

A single precision real constant may be expressed in one of two forms.

In the basic form, a real constant is expressed as a string of one to nine significant decimal digits with one decimal point preceding, imbedded in, or following the digits. The constant may be signed or unsigned.

The second form consists of a real constant followed by a decimal exponent. The power of ten is expressed by appending the letter E followed by a signed or unsigned integer to the real constant. A decimal point does not need to be included in the real part of this form.

The approximate magnitude of a real constant must be between 10^{-38} and 10^{38} . A real constant occupies one word (four bytes) of storage:

S	Characteristic	Mantissa
1	2 9	10 36

A real constant is approximate if it contains a fractional part, since the value is stored in binary floating-point form. The following are valid real constants:

0.0
 .0
 1.
 -15.07
 2.0E2 (means 200.0)
 2E2 (means 200.0)
 0.00095
 4.0E2 (means 400.0)
 4.0E+2 (means 400.0)
 4.0E-2 (means 0.04)

2.2.1.2.2. Double Precision

A double precision real constant may have up to 18 significant digits and may have an approximate magnitude in the range of 10^{-308} to 10^{308} .

Generally, double precision real constants are specified with a decimal point and 1 to 18 significant digits. A double precision constant must contain an exponent which consists of the letter D followed by a signed or unsigned integer. The D has the same meaning for double precision as the E has for single precision.

This form of constant occupies two words (eight bytes) of storage:

S	Characteristic	Mantissa
1	2 12	13 36

Mantissa (continued)	
1	36

The following are acceptable double precision constants:

0.0D0 (means 0)
 1.0D0 (means 1.0)
 1D0 (means 1.0)
 16.9D+1 (means 169.0)
 +8.897D+10
 -1750.D+19
 123.4567891D0
 .1234567891D3 (The values of these last two constants are equal.)

2.2.1.3. Complex Constants

A complex constant is written as a pair of real or integer constants. The general form is:

$$(r,i)$$

where r is the real part and i is the imaginary part. Both parts may be either single precision (integer or real) or double precision. A single precision complex constant requires two consecutive words (eight bytes) of storage, with the real part occurring first. Double precision complex constants are formed by a pair of double precision real constants and require four words (16 bytes) of storage (see 2.2.1.2). Both parts of a complex constant must have the same precision. The following are valid complex constants:

```
(0.0,1.0)
(2,3)
(3426.78,293.6)
(4.12E2,6.5)
(4.12E-2,6.5E+3)
(4.12E-10,6.5E-3)
(4.12D-10,6.5D-3) (double precision)
```

A single precision complex constant specified with two integer constants will be represented internally (in storage) as two single precision real numbers.

2.2.1.4. Logical Constants

A logical constant specifies a logical value which is either "true" or "false." The only valid logical constants are:

```
.TRUE.
.FALSE.
```

2.2.1.5. Character Constants

A character constant is a string of characters which may include any of the ASCII characters. Character constants are sometimes referred to as literals or Hollerith constants. They may be indicated in one of two ways:

- The string may be enclosed in apostrophes (referred to as the literal form). Two apostrophes in succession represent a single apostrophe in the text. The following are valid character constants:

```
'ABCD'
'123ABC$! !'
'THAT''S'
'xyz' (nonstandard because it contains lowercase letters)
```

In the third example, the code produced would be:

```
THAT'S
```

- The second method for defining a character constant is to precede the constant with n H where n is the number of characters in the string (referred to as the Hollerith form). Valid character constants of this form are:

```
3HWOW
6H THAT'S
4HA1+$
27HUPPER and lowercase letters
```

Each character requires one 9-bit byte (one-fourth of a word) of storage. A character constant consisting of n characters requires the integer value of $((n+3)/4)$ words of storage for its representation. All character data is represented in the full ASCII form. The 9-bit representation of ASCII consists of one 0-bit followed by the 8-bit ASCII code. The maximum size of a character constant is 511 characters.

If a character constant is split between two or more lines in the source program, it is important to remember that the constant extends through column 72 of the first line and begins again in column 7 of the following continuation line.

2.2.1.6. Octal and Fielddata Constants

Octal and Fielddata constants are allowed only in the data lists of specification statements and as input to namelists.

An octal constant is expressed by the letter O followed by 1 to 12 octal digits.

A Fielddata constant is written as a quoted or Hollerith literal immediately followed by the letter F.

2.2.2. Symbolic Names

A symbolic name consists of one to six of the following characters:

- uppercase and lowercase letters
- numerics (0, 1, 2, . . . , 9)
- currency symbol (\$)

The first character of each symbolic name must be alphabetic.

The programmer is entirely free in the choice of words for symbolic names. Keywords such as GO TO, READ, FORMAT, etc., are not considered reserved words and may therefore be used as symbolic names or as parts of symbolic names. However, when GO TO, READ, FORMAT, etc., are used as statement keywords, they are not considered to be symbolic names. The same applies to sequences of characters which are format field descriptors such as I3 (see 5.3.1). The use of keywords as variables, function names, or subroutine names is not recommended since such usage makes programs difficult to read.

In general, it is advisable not to use the currency symbol (\$) in function and subroutine names. This avoids conflicts with entry names in the ASCII FORTRAN library and the Series 1100 Operating System relocatable library.

Symbolic names containing lowercase alphabets are identical to symbolic names containing uppercase alphabets. Thus, the name xxx is identical to the name XXX.

2.2.2.1. Uniqueness of Symbolic Names

In a program unit, a symbolic name, perhaps qualified by a subscript, identifies a member of one (and only one) of the following fourteen classes unless noted otherwise:

- An array and the elements of that array
- A scalar variable
- A statement function
- An intrinsic function
- An external function
- An internal function
- An external subroutine
- An internal subroutine
- An external procedure which cannot be classified as either a subroutine or a function in the program unit in question.
- A common block
- A constant defined in a PARAMETER statement
- A NAMELIST name
- A program bank name
- A main program name

A common block or bank name in a program unit may also be the name of any local entity other than a parameter constant, intrinsic function, or local variable that is also an external function in a function subprogram.

A main program name is global to the executable program and must not be the same as the name of an external subprogram in the same executable program.

A FUNCTION subprogram name must also be a variable name in the FUNCTION subprogram.

Once a symbolic name is used as a program bank name, a FUNCTION subprogram name, a SUBROUTINE subprogram name, or an external procedure name in any unit of an executable program, no other program unit of that executable program may use that name to identify another member of any of these four classes.

Note that the source input to the compiler may contain more than one program unit (see 10.2). Each compilation produces one relocatable element regardless of the number of program units. There are internal and external program units. The only variables shared between external program units are parameters passed between them and variables defined in common blocks. Local variable names in one external program unit have no relationship to the local variable names in another external unit. Internal subprograms may have their own local variables and, in addition, may directly access the variables of their external program unit (see 7.4.2 and 7.4.3).

For example, suppose the source input consists of a main program and an external subroutine. The symbolic name *A* may be used as a local scalar variable in the main program and also as a local array in the subroutine. The two *A*'s have no relationship to each other; each *A* has its own storage location.

2.2.2.2. Data Types of Symbolic Names

A symbolic name representing a function (except intrinsic functions), a variable, or an array must have only a single data type in a program unit. This type is associated with usage of the symbolic name throughout the program unit.

A symbolic name may be one of five types:

- Integer
- Real (single precision or double precision)
- Complex (single precision or double precision)
- Logical
- Character

There are no literal symbolic names (that is, 'CON' versus CON). Literals can only be represented as constants. A constant automatically defines its own type by the form of its appearance.

The data type of a symbolic name can be explicitly declared by the programmer or implied by its first letter. The data type of a function determines the type of the datum it supplies to the expression in which it appears.

The data type of an array element name is the same as that of its array name.

2.2.2.2.1. Implied Declaration Via the Name Rule

Each variable, array, or function (except intrinsic functions) has a type implied by the name rule. This convention determines the type based on the first character of the symbolic name:

- If the first character of the variable name is one of the six alphabets I, J, K, L, M, N, the variable is type integer.
- If the first character of the name is any other alphabetic character, the type is real, single precision.

This convention is the traditional FORTRAN method of implicitly specifying the type of a variable as being either integer or real. Variables defined using this convention are of standard length. Double precision real, complex (single precision and double precision), logical, and character types cannot be declared by the name rule.

2.2.2.2.2. Implied Declaration Using the IMPLICIT Statement

The IMPLICIT statement overrides the type established by the name convention.

The IMPLICIT statement allows the programmer to specify the data type and length to be associated with specified initial letters. The IMPLICIT statement can be used to specify all types of variables (integer, real, character, complex, and logical) and to indicate length specifications (precision).

See 6.3.1 for details on the IMPLICIT statement.

2.2.2.2.3. Explicit Declaration

Declaration by a type statement overrides the name rule and type specifications of IMPLICIT statements.

Explicit specification statements differ from the first two ways of specifying the type of variable, in that an explicit specification statement declares the type of a particular variable by its name rather than that of a group of variable names beginning with a particular letter.

A symbolic name can be assigned a specific type and a specific length using an explicit specification statement (see 6.3.2).

2.2.2.3. Variables

A variable is a symbolic name that identifies a single storage area in which its current value can be found. The size of this storage area and the type of value to be put in that area are determined by the type declared implicitly or explicitly for that variable. The allowed types for variables are: integer, single or double precision real, single or double precision complex, character, or logical.

The contents of the storage area represented by the variable is undefined prior to assignment of its first value. Any reference to the variable before its assignment or initialization is undefined unless the reference assigns it a value. The value of a variable may be defined by a DATA statement, by an input/output statement, by an assignment statement, by its use as an argument in a subprogram reference, by its use in a DO statement, or by association in a COMMON or EQUIVALENCE statement. The data type of a variable corresponds to the type of data the variable represents. Thus, an integer variable represents integer data, a real variable represents real data, etc. A variable occupies the same number of storage locations as a constant of the same type.

Once the type of a variable has been defined, it may be assigned different values of this type as desired within the program. This changeability distinguishes it from a constant. The following is an example of variables in use:

```
SUM = TOTAL1 + TOTAL2
```

The symbol "=" indicates that the value of the evaluated right-hand expression is to be assigned to the variable on the left. SUM, TOTAL1, and TOTAL2 are variables. The sum of the values represented by TOTAL1 and TOTAL2 is to be stored in the storage area represented by SUM at execution time. As with all expressions, TOTAL1 and TOTAL2 must have been previously assigned values to obtain a defined (predictable) value for SUM.

Variables and constants are sometimes referred to as scalars.

A choice of appropriate variable names can aid documentation of a program and make the program more readable.

2.2.2.4. Arrays and Subscripts

An array is an indexed set of variables identified by a symbolic name (the array name). The members of the set represented by the array are referred to as array elements.

The array name together with its position in the array comprise the unique identification of an array element. An element's position in the array is indicated by a parenthesized expression, known as a subscript, following the array name. The subscript must be present to ensure proper identification of the array element. For example:

A(1)

identifies the first element in array A (if the lower dimension bound for A is 1; see 2.2.2.4.1). Each appearance of an array name must be with its qualifying subscripts except in the following cases:

- In a DIMENSION statement. Note that although the dimension declarator may resemble an array's subscripts they are not the same.
- In a COMMON statement.
- In a type statement.
- In an EQUIVALENCE statement.
- In a DATA statement.
- In a list of arguments for a reference to a subprogram.
- In a list of dummy arguments.
- In a list of an input/output statement, if the array is not an assumed-size array.
- As a unit identifier for an internal file in an input/output statement, if the array is not an assumed-size array.
- As the format identifier in an input/output statement, if the array is not an assumed-size array.
- In a SAVE statement.

For the preceding cases, the array name without the qualifying subscripts identifies the entire sequence of elements of the array except for the EQUIVALENCE statement (see 6.4).

As with variables, an array and its elements have no defined value until they have been assigned a value in some way.

2.2.2.4.1. Array Declaration

Like a variable, an array may be integer, single or double precision real, single or double precision complex, character, or logical. The type of the array is also that of its data elements. This type may be declared implicitly or explicitly.

An array can be declared in a DIMENSION statement, a COMMON statement, or type statement. The form of the array declaration part is:

$$a(d [, d] . . .)$$

where a is a symbolic name and d is a dimension declarator of the form:

$$[d1 :] d2$$

The dimension $d1$ is the optional lower dimension bound and is assumed to be 1 if omitted; $d2$ is the upper dimension bound. The number of dimensions of an array is the number of dimension declarators in the array declaration. The minimum number of dimensions is one, and the maximum is seven.

The lower and upper dimension bounds are arithmetic expressions in which all constants, parameter constants, and variables must be type integer. The upper dimension bound of the last dimension may be an asterisk (*) in an assumed-size array (see 2.2.2.4.3). A dimension bound expression must not contain a function or array element reference. Integer variables may appear in dimension bound expressions only for adjustable arrays.

A variable or parameter constant appearing in a dimension bound expression which is not of default integer type must be specified as integer by an IMPLICIT statement or a type statement prior to its use in the dimension bound expression.

The following are valid array declarations:

```
DIMENSION ARRAY1(2,3), VARIED(-10:L)
DOUBLE PRECISION ARRAY2(15), LIST2(K:2*K, 100)
COMPLEX RTAB(0:10, 0:20, *), USIZ(*)
COMMON WANTED(3,4,5)
```

2.2.2.4.2. Value of Dimensions

The value of either dimension bound may be positive, negative, or zero. The value of the upper dimension bound must be greater than or equal to the value of the lower dimension bound. If only the upper dimension bound is specified, the value of the lower dimension bound is one. An upper dimension bound of an asterisk is always greater than or equal to the lower dimension bound.

2.2.2.4.3. Constant, Adjustable and Assumed-Size Arrays

Arrays dimensioned with only integer constant expressions are known as constant dimensioned arrays. An adjustable array has one or more integer variables specified in its dimension declarator. The integer variables must be either formal parameters or common block variables which must be defined at the time of execution of the reference to the subprogram containing the adjustable array. An assumed-size array is a constant or adjustable array except that the upper dimension bound of the last dimension is an asterisk.

An adjustable array or assumed-size array can only appear in functions or subroutines, and the array name must appear in the formal parameter list. An array name in a formal parameter list is called a dummy array. The actual argument corresponding to the dummy array name may be an array name, array element, or array element substring. The number and values of the dimensions need not be the same in both the calling and called routines. Storage for the dummy array is not allocated in the subprogram and the dummy array can not assume more storage than what its corresponding actual argument has been allocated.

The following are examples of the three types of arrays:

```
SUBROUTINE SUB1(X,Y,Z,I)
DIMENSION X(1900:1978)      @ constant dimensioned array
REAL Y(1:2*I)              @ adjustable dimensioned array
CHARACTER*8 Z(*)           @ assumed-size array
```

2.2.2.4.4. Actual and Dummy Arrays

Arrays dimensioned with the array name not appearing in a formal parameter list are known as actual arrays. An actual array has a determinable size and is allocated storage. Actual arrays must also be constant dimensioned arrays. An actual array may be declared in a DIMENSION, COMMON, or type statement.

A dummy array is an array in which the array name appears in a formal parameter list. A dummy array may be either a constant, adjustable, or assumed-size array. A dummy array is permitted in a DIMENSION statement or a type statement, but not in a COMMON statement. A dummy array may appear only in a function or subroutine subprogram.

2.2.2.4.5. Array Element Reference

An array element is identified by the array name followed by a parenthesized subscript expression representing its position within the array. The expression consists of one to seven subscripts separated by commas, forming a list which is enclosed in parentheses. The number of subscripts must correspond to the number of dimensions specified when the array dimensions were declared.

Each subscript must be an arithmetic expression which yields an integer, real, or double precision real value. When the subscript is evaluated, and converted to integer if necessary, the value may be negative, positive, or zero. The value must not exceed its corresponding array dimension. The user is cautioned that the integer subscript result of a real subscript expression may not be what is expected because of the approximate nature of real numbers (see 2.2.1.2.1). If the upper dimension bound is an asterisk, the value of the corresponding subscript expression must be such that the subscript value does not exceed the size of the actual array. Subscript references are unrestricted in form and may include array element or function references.

The expression B(3,2) refers to the element in the third row, second column of array B. The expression C(3/3,SQRT(4.),4) refers to the element in the second row, fourth column, of the first plane of array C.

2.2.2.4.6. Location of Elements Within an Array

The elements of each array are stored in column-major order. This means the array element with lowest subscripts is stored in the lowest storage position and those with higher subscripts (leftmost subscripts increasing most rapidly) are stored in subsequent storage positions. For example, if array B is declared:

```
DIMENSION B(3,3)
```

The array elements of B would be stored in ascending storage positions in the following order:

B(1,1)
B(2,1)
B(3,1)
B(1,2)
B(2,2)
B(3,2)
B(1,3)
B(2,3)
B(3,3)

2.2.2.5. Character Substrings

A character substring is a contiguous portion of a character variable or array element and is of type character. A character substring is identified by a substring name and may be assigned values and referenced.

The forms of a substring name are:

$v([e_1] : [e_2])$

$a(s[, s] . . .) ([e_1] : [e_2])$

where:

v is a scalar character variable name.

$a(s[, s] . . .)$ is a character array element name.

e_1 and e_2 are integer expressions called substring expressions.

The value e_1 specifies the leftmost character position of the substring, and the value e_2 specifies the rightmost character position.

For example, A(2:4) specifies characters in positions two through four of the scalar character variable A. B(4,3)(1:6) specifies characters in positions one through six of character array element B(4,3).

The values of e_1 and e_2 must satisfy the following relational expression:

$$1 \leq e_1 \leq e_2 \leq len$$

where len is the length of the scalar character variable or array element. If e_1 is omitted, a value of one is implied. If e_2 is omitted, a value of len is implied. Both e_1 and e_2 may be omitted. For example, the form $v(:)$ is equivalent to v , and the form $a(s[s] \dots)(:)$ is equivalent to $a(s[s] \dots)$.

The length of a character substring is $e_2 - e_1 + 1$. Therefore, the expression $v(e_1:e_2)$ is equivalent to the expression SUBSTR($v, e_1, e_2 - e_1 + 1$). See 7.3.2.2 for a description of the SUBSTR pseudo-function. Note that the SUBSTR pseudo-function is not allowed in certain places in which a substring is allowed (for example, DATA and EQUIVALENCE statements).

Example:

```
CHARACTER*4 C1, C2(2,2)
C
C - THE FOLLOWING STATEMENTS ARE EQUIVALENT
C
C2(1,1)(1:J) = C1(3:4)
SUBSTR(C2(1,1),1,J-1+1) = SUBSTR(C1,3,2)
```

2.2.3. FORTRAN Expressions

An expression is a group of one or more elements and operators which is evaluated to form a single value within a statement.

Four kinds of FORTRAN expressions result from combinations of operands and operators. They are:

- Arithmetic
- Character
- Logical
 - using logical operators
 - using relational operators
- Typeless

The value of an arithmetic expression is always integer, real (single or double precision), or complex (single or double precision). A character expression always yields an ASCII character string of length one or greater. The value of a logical expression is always `.TRUE.` or `.FALSE.`, while that of a typeless expression is always a 36-bit string.

2.2.3.1. Arithmetic Expressions

A simple arithmetic expression consists of one or two operands and an arithmetic operator.

2.2.3.1.1. Arithmetic Operators

The following set of arithmetic operators is included in the language:

<u>Operator</u>	<u>Operation</u>
+	Binary addition or unary plus
-	Binary subtraction or unary minus
*	Multiplication
/	Division
**	Exponentiation

2.2.3.1.2. Formation of Arithmetic Expressions

A simple arithmetic expression is $[a] \text{ op } b$ where a and b are variables and op is an arithmetic operator. The form of a more complex arithmetic expression is:

$$[[\text{sign}] \text{ term }] \text{ op } \text{ term}$$

Table 2-2 gives the form of *term*.

Table 2-2. Composition of an Arithmetic Term

Entity	Definition	Examples
term	A term is composed of a product, term + product, or term - product.	A*B A+B+C*D A+B-C*D
product	A product is a factor, a product*factor, or a product/factor.	A**A A*B*A**B A/B*A**B
factor and power	A factor is a primary, or a primary**power. Power has the form of a factor.	A A**A**A
primary	A primary is an unsigned constant, variable reference, array element reference, function reference, or an arithmetic expression in parentheses.	A 10 (A+B) ARRAY(I) FUNC(A,B)

As indicated in Table 2-2, an arithmetic expression may not contain two consecutive operators.

Examples of valid arithmetic expressions are:

```
X(2)
-2
-(-2)
4**2
-(-2)+4**2-16
```

where X is an arithmetic array or function.

2.2.3.1.3. Evaluation of Arithmetic Expressions

If an arithmetic expression includes more than one arithmetic operator, evaluation of that expression is performed based on the following hierarchy:

<u>Operation</u>	<u>Example</u>	<u>Order of Performance (Rank)</u>
Expressions in parentheses	(<i>e</i>)	1 st
Function evaluation	SQRT(<i>e</i>)	2 nd
Exponentiation	2**3	3 rd
Multiplication and division	2*3 2/3	4 th
Addition and subtraction or their unary operations	2+3 -2	5 th

Expressions are evaluated, with one exception, in a left-to-right fashion. If a subexpression contains the form:

$$a \text{ } op_1 \text{ } b \text{ } op_2 \text{ } c$$

the part $a \text{ } op_1 \text{ } b$ is evaluated first as long as op_1 has a greater or equal rank with respect to op_2 as described by the preceding table. The one exception is the form:

$$a \text{ ** } b \text{ ** } c$$

The part $b \text{ ** } c$ is evaluated first.

Examples of equivalent expressions:

$$a - b * c ** d \text{ is equivalent to } a - (b*(c**d))$$

$$a - b * c * d \text{ is equivalent to } a - ((b*c)*d)$$

$$a ** (b - c) * d \text{ is equivalent to } (a**(b - c))*d$$

2.2.3.1.4. Type Rules for Arithmetic Expressions

Table 2-3 is used in evaluating an arithmetic expression $a \text{ } op \text{ } b$.

Table 2-3. Length and Type of Result for Arithmetic Expressions

Type of Left Operand (a)	Type of Right Operand (b)				
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	COMPLEX*16
INTEGER	INTEGER	REAL	REAL	COMPLEX	COMPLEX*16
REAL	REAL	REAL	REAL	COMPLEX	COMPLEX*16
DOUBLE PRECISION	REAL	REAL	REAL	COMPLEX*16	COMPLEX*16
COMPLEX	COMPLEX	COMPLEX	COMPLEX*16	COMPLEX	COMPLEX*16
COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16

The length of the operands and results are:

REAL and INTEGER – one word (4 ASCII characters)

DOUBLE PRECISION and COMPLEX – two words (8 ASCII characters)

COMPLEX*16 – four words (16 ASCII characters)

The data type of a unary operation is the same as that of its operand.

2.2.3.2. Character Expressions

A character expression has a value representing a sequence of one or more ASCII characters. It may be a character constant (literal), character variable, character array element, character substring (see 2.2.2.5), character function reference, or another character expression enclosed in parentheses. A character expression may also be two or more of the aforementioned items combined by using the binary concatenation operator "//".

The binary concatenation operator "//" is the only character operator in FORTRAN. It is used in the following manner, where e_1 and e_2 are character expressions:

$$e_1 // e_2$$

Assume e_1 is a character expression of length m and e_2 is a character expression of length n . The result of the expression is a character expression of length $m + n$. The first m characters of the resulting expression are those of e_1 . The remaining n characters are those of e_2 .

For example, the expression:

'TWIST' // 'ANDTURN'

results in the value 'TWIST AND TURN'.

The expression:

'TYPE' // '123'

results in the value 'TYPE123'.

For syntactic compatibility with other FORTRAN processors, '&' is also allowed as a concatenation operator.

FORTRAN 77 does not allow a character item with a length of * to appear as a concatenation operand in an expression in a subprogram argument or an I/O list. However, ASCII FORTRAN does not have this restriction.

2.2.3.3. Logical and Relational Expressions

A logical expression has the value .TRUE. or .FALSE..

Logical expressions can be divided into two groups based on the type of data handled by their operators. These two groups are:

- Pure logical
- Relational

Pure logical expressions allow only logical operands.

Relational expressions allow logical, integer, real (single or double precision), complex (single or double precision), typeless, and character expressions as operands.

The basic form of a logical expression is similar to that of an arithmetic expression.

2.2.3.3.1. Logical Operators

Logical operators can be divided into two groups: pure logical (Table 2-4) and relational (Table 2-5). The operands (e_i) for pure logical operators may be of logical or typeless type (see 2.2.3.4).

Table 2-4. Pure Logical Operators

Operator	Usage	Explanation
.NOT.	.NOT. e_1	The expression has the logically opposite value of the expression e_1 .
.AND.	e_1 .AND. e_2	If both e_1 and e_2 have the value of .TRUE., the expression has the value .TRUE. ; otherwise it has the value .FALSE.
.OR.	e_1 .OR. e_2	If either, or both, e_1 or e_2 has the value .TRUE, then the expression has the value .TRUE. ; otherwise the expression has the value .FALSE.
.EQV.	e_1 .EQV. e_2	If both e_1 and e_2 have the value of .TRUE. or both have the value of .FALSE., then the expression has the value of .TRUE., otherwise it has the value of .FALSE.
.NEQV.	e_1 .NEQV. e_2	If e_1 has the value of .TRUE. and e_2 has the value of .FALSE., or if e_1 has the value of .FALSE. and e_2 has the value of .TRUE., then the expression has the value of .TRUE.; otherwise the expression has the value of .FALSE.

Table 2-5. Relational Operators

Operator	Usage	Explanation
.GT.	e_1 .GT. e_2	True if e_1 is greater than e_2 .
.GE.	e_1 .GE. e_2	True if e_1 is greater than or equal to e_2 .
.LT.	e_1 .LT. e_2	True if e_1 is less than e_2 .
.LE.	e_1 .LE. e_2	True if e_1 is less than or equal to e_2 .
.EQ.	e_1 .EQ. e_2	True if e_1 is equal to e_2 .
.NE.	e_1 .NE. e_2	True if e_1 is not equal to e_2 .

The operands (e_i) for relational operators may be integer, real (single or double precision), complex (single or double precision), typeless, or character. A complex operand is permitted only when the relational operator is .EQ. or .NE.. However, if one of the e_i is of type character, the other must also be of type character. The comparison of a double precision value and a complex value is permitted.

2.2.3.3.2. Formation of Logical Expressions

A logical expression consists of logical operators and allowable operands. It produces a logical value of `.TRUE.` or `.FALSE.`.

A logical expression has one of two forms:

- logical term
- logical expression `.OR.` logical term

Table 2-6 defines a logical term.

Table 2-6. Composition of Logical Term

Entity	Definition	Examples
logical term	A logical term is composed of a logical factor, or logical term <code>.AND.</code> logical factor.	L <code>.AND.</code> <code>.NOT.</code> L
logical factor	A logical factor can be either a logical primary or <code>.NOT.</code> logical primary.	<code>.NOT.(A .GT. B)</code>
logical primary	A logical primary is a logical constant, logical variable reference, logical array reference, logical function reference, relational expression, or logical expression enclosed in parentheses.	L <code>.TRUE.</code> LARRAY(I) A <code>.GT.</code> B (A <code>.OR.</code> B)

A relational expression is of the form:

$$e_1 \text{ relop } e_2$$

where *relop* is a relational operator and e_i are expressions. If one of the e_i is of type character, the other must also be of type character.

2.2.3.3.3. Evaluation of Logical Expressions

If a logical expression contains more than one logical operator, evaluation of that expression is performed using the following hierarchy:

<u>Operator</u>	<u>Rank</u>
Relationals	1 st (evaluated first)
<code>.NOT.</code>	2 nd
<code>.AND.</code>	3 rd
<code>.OR.</code>	4 th
<code>.EQV.</code> , <code>.NEQV.</code>	5 th (evaluated last)

If more than one relational operator exists, they are performed in left-to-right order. The same applies to multiple appearances of the same operator.

In relational expressions with character operands, character relations are performed left to right using the ASCII collating sequence. If one of the two expressions needs to be extended to make it equal in length to the other, it is extended on the right using blanks.

Assuming A has the value .TRUE. and B the value .FALSE., examples of logical expressions and their values are:

<u>Expression</u>	<u>Evaluation</u>
A .AND. B	.FALSE.
A .AND. .NOT. B	A .AND. .TRUE. .TRUE.
3.LT.4.OR.B	.TRUE. .OR.B .TRUE.
.NOT. A .AND. B	.FALSE. .AND. B .FALSE.
.NOT.3.LE.4.OR.B	.NOT. .TRUE. .OR. B .FALSE. .OR.B .FALSE.
A .NEQV. B	.TRUE.

2.2.3.4. Typeless Expressions

A typeless expression is a 1-word (36-bit) string which may appear alone or in an expression together with an integer, real, character, logical, or another typeless expression. A typeless expression may be composed of any valid FORTRAN constants or variables and one of the typeless functions.

2.2.3.4.1. Typeless Functions

The following primitive typeless functions are permitted in ASCII FORTRAN where e_1 is any single word expression:

<u>Expression</u>	<u>Explanation</u>
AND (e_1, e_2)	Bit by bit logical product
OR (e_1, e_2)	Bit by bit logical sum
XOR (e_1, e_2)	Bit by bit exclusive OR
BOOL (e_1)	Treat e_1 as typeless
COMPL (e_1)	Bit by bit complement

The results of the above functions, found by treating a 0 bit in the sense of "false" and a 1 bit in the sense of "true," are typeless expressions.

2.2.3.4.2. Evaluation of Typeless Expressions

If an expression consists only of typeless expressions and arithmetic operators, or of typeless expressions and relational operators, the typeless expressions (bit strings) are treated as though they were integer values.

If an expression consists only of typeless expressions and logical operators, the typeless expressions are treated as though they were logical values, in which case only the rightmost bit of the expression is significant.

If an expression (or part of an expression) is of the form $e_1 \text{ op } e_2$ where one of the e_i is a typeless quantity and the other expression is of type integer, real, character, or logical, then assuming op is a suitable operator for comparing the two entities, the typeless quantity is taken to be integer, real, character, or logical, respectively, without being converted to that type. If an expression is of type character and less than four characters in length (one word), the result is undeterminable.

A typeless expression must not be combined with a complex or double precision expression. The order of expression evaluation is significant when typeless expressions are involved. For example, the expression:

`(integer+typeless)+real`

is not the same as the expression

`integer+(typeless+real)`

The typeless entity is taken to be integer in the first case, real in the second.

2.2.3.5. Hierarchy of Operators

The following hierarchy should be used to determine the evaluation order of subexpressions involving the operators mentioned in subsections of 2.2.3.

<u>Kind</u>	<u>Operation</u>	<u>Rank</u>
Parenthesized	All	1 st (evaluated first)
Functions	All	2 nd
Arithmetic	Exponentiation (**)	3 rd
	Multiplication and division (* and /)	4 th
	Addition and subtraction (+ and -) and unary operation	5 th
	Concatenation (// and &)	6 th
Logical	Relational comparisons (.GT., .GE., .LT., .LE., .EQ., .NE.)	7 th
	.NOT.	8 th
	.AND.	9 th
	.OR.	10 th
	.EQV., .NEQV.	11 th (evaluated last)

Note the following:

- Expressions are evaluated left-to-right, except where the hierarchy dictates otherwise, or in the case of successive exponentiation operators (see 2.2.3.1.3).
- Logical expressions may not require that all parts be evaluated. For example, if *A* is a logical variable whose value is `.TRUE.` and *LEG* is a logically valued function, the expression `A .OR. LEG(x)` may not result in a call to the function *LEG*. Since *A* has a true value, the value of the expression is already determinable.

2.2.4. General Statement Form

The general form of all FORTRAN statements is:

1	5	6	7	72	73	80
<i>label</i> or blank			<i>statement</i>			nonprocessed documentation

Column 6 is used for a continuation character, if the line is to be a continuation line (see 2.2.6).

Note that any text (except comments) extending beyond column 72 is moved over three columns and a space-period-space is inserted before the source line is printed.

2.2.5. Comment Line

Any line with the characters `C` or `*` in column 1 is treated as a comment and not as a FORTRAN statement. A line that contains only blank characters in columns 1 through 72 will be treated as a comment line. Comments are included with listings and in the symbolic element for the convenience and information of the programmer, but they are not compiled as FORTRAN statements.

Comment lines may appear anywhere within a program unit. FORTRAN 77 does not allow comment lines following an `END` statement.

2.2.6. Continuation Line

Any character other than a blank or zero appearing in column 6 of a FORTRAN statement signifies that this line is a continuation of the prior FORTRAN statement. Columns 1 through 5 of a continuation line may contain only blank characters.

Each FORTRAN statement may contain **at least 20 lines but not more than 1320 significant characters.** Blanks are not significant characters except in character literals and Hollerith fields. For instance, a FORTRAN statement containing only significant characters could be continued for a maximum of 19 lines (assuming lines with statement coding in columns 7 through 72). **A FORTRAN statement may contain more than 20 lines depending upon the number of significant characters.**

Each continuation line must be preceded by an initial line or another continuation line.

2.2.7. Inline Comment

To retain compatibility with SPERRY UNIVAC FORTRAN V, the user may attach a comment to any line by placing a master space character (@) before the comment. An inline comment may not start in columns 1 through 6.

Example:

```
X = 5.0      @ INITIALIZE X
```


3. Assignment Statements

3.1. General

Assignment statements are the basic mechanism by which the result of an expression is stored (and therefore, saved) in a variable for future reference. Many types of conversions may be induced during evaluation of the expression before the final result is stored.

There are four types of assignment statements:

- Arithmetic assignment, for storing the result of an arithmetic expression, **character constant, or Hollerith constant** in a numeric (integer, real, or complex) variable or array element.
- Character assignment, for storing the result of a character expression into a character variable, character array element, or character substring.
- Logical assignment, for storing the **.TRUE.** or **.FALSE.** result of a logical expression into a logical variable or logical array element.
- Statement label assignment (ASSIGN statement), for storing the location of a statement label in an integer variable. The unique form of this type of assignment statement unambiguously differentiates statement labels from numeric constants.

The equal sign (=) is the usual assignment operator.

Special rules for conversion, size determination, etc., apply in many cases. These are cited in the discussion of each type of assignment statement.

3.2. Arithmetic Assignment Statement

Purpose:

The arithmetic assignment statement is used to transfer a numeric value, the result of an arithmetic expression, to a variable or array element of type INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or **COMPLEX*16**.

Form:
$$v [, v] \dots = e$$
where:

- v is a variable, array element, or pseudo-function (see 7.3.2).
- e is an arithmetic expression, character constant, Hollerith constant.

Description:

The result of the expression is stored in the target variables which appear to the left of the assignment "=" operator. If the result of the expression differs in type from the target variables, conversions will be performed in most cases, as indicated in the Table 3-1.

Caution should be exercised when using user-defined functions as array subscripts or pseudo-functions arguments.

The sequence of execution is as follows:

1. Subscripts or pseudo-function arguments of the target variables are evaluated.

The order of this evaluation is not set. Therefore, side effects, which are hidden changes in variables other than the one being evaluated, may produce results dependent on the sequence of object code. The results of such side effects are not defined. Programs containing such constructs may therefore produce different results when executed on different FORTRAN systems. See the examples at the end of this subsection.

2. The expression is evaluated. Beware of side effects which may be present since their effect is undefined.
3. For each target variable, the result of the expression is converted, if necessary, and stored into the variable. The order of this assignment is not specified. A character constant or Hollerith constant is stored to an arithmetic data item with no conversion being performed regardless of the length of the literal item or the data item into which it is being stored.

Table 3-1. Conversions for Arithmetic Assignment

Type of Expression Result	Type of Target				
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	COMPLEX*16
INTEGER	Store.	Float. Store.	Double Float. Store.	Float, store to real part. Store 0 as imaginary part.	Double Float. Store to real part. Store 0 as imaginary part.
REAL	Fix. Store.	Store.	Extend to double. Store.	Store to real part. Store 0 as imaginary part.	Extend to double. Store to real part. Store 0 as imaginary part.
DOUBLE PRECISION	Fix. Store.	Contract to single. Store.	Store.	Contract to single, store to real part. Store 0 as imaginary part.	Store to real part. Store 0 as imaginary part.
COMPLEX	Fix, store real part. Ignore imaginary part.	Store real part. Ignore imaginary part.	Extend real part to double; store. Ignore imaginary part.	Store.	Extend both parts to double. Store.
COMPLEX*16	Fix, store real part. Ignore imaginary part.	Contract real part to single store. Ignore imaginary part.	Store real imaginary imaginary part.	Contract both parts to single, store.	Store.
LOGICAL or CHARACTER (nonliteral)	Error.	Error.	Error.	Error.	Error.

Examples:

```
C      A=B+C
      The value of the expression B + C is stored in A.

C      INDX=INDX+1
      The value of the expression INDX+1 is stored in INDX.

C      X=CSIN(W*PI+THETA)-ORIGIN
      The value of the expression is stored in the variable X.

C      N=1
      The constant one is stored in N.

C      END
      The following program sequence may depend upon side effects
C      because of the order of evaluation and optimization.
C      Since side effect dependence is forbidden, the sequence
C      may not produce the results intended by the programmer.

      .
      .
      .
      DIMENSION A(10,10,10)
      COMMON/WATCH/MIGHT,CHANGE
      SUBJCT = 1
      MIGHT = 2
      CHANGE = 4
      A(MIGHT, FC(SUBJCT),CHANGE) = D
      .
      .
      .
      END

C      FUNCTION FC(A)
      COMMON/WATCH/MIGHT,CHANGE
      MIGHT = MIGHT + 1
      CHANGE = CHANGE + 1
      FC = MIGHT + CHANGE - A
      .
      .
      .
      END

C      The example above shows the introduction of side effects during
C      evaluation of the function FC due to the COMMON properties of
C      MIGHT and CHANGE. ASCII FORTRAN presently evaluates
C      A(MIGHT,FC(SUBJECT),CHANGE) to A(3,7,5). However, the user
C      is warned not to rely upon side effects because a change in
C      compiler evaluation of the code could change the result.
```

3.3. Character Assignment Statement

Purpose:

The character assignment statement is used to transfer the result of a character expression to a character variable, character array element, or character substring.

Form:

$$c [: c] \dots = ce$$

where:

c is a character variable, character array element, character substring, or pseudo-function (see 7.3.2).

ce is a character expression.

Description:

The result of the character expression is stored in the target variables which are on the left of the assignment ("=") operator.

If the expression involves character variables, character array elements, character substrings, or pseudo-functions, the target c must be type character since no type conversions will be performed. If the expression is a character literal or Hollerith string, the target variables can be of any type; no conversions will be performed.

If the length of the character string result of the expression differs from the size of the target variables, the string will be truncated or padded with blanks as indicated in Table 3-2 (L_1 is the length of the target in characters; L_2 is the length of the result of the character expression in characters):

Table 3-2. Conversions for Character Assignment

Condition	Action
$L_1 < L_2$	The leftmost L_1 characters of the expression result are stored in the target; the remaining characters of the expression result are ignored.
$L_1 = L_2$	The expression result is stored in the target.
$L_1 > L_2$	The expression result is stored in the leftmost L_2 characters of the target; the remaining rightmost L_1 minus L_2 characters are filled with blanks.

The sequence of execution is the same as for arithmetic assignment statements.

Examples:

```
CHARACTER*12 CSFMSG, COMMND
CSFMSG='@' // COMMND
C      The character string result of the concatenation expression
C      '@' // COMMND is stored in the character variable CSFMSG.
```

```
CHARACTER BLANK*1
```

```
BLANK=' '
C      The single blank is stored in character variable BLANK.
END
```

```
CHARACTER*4 S2(10,10)
```

```
S2(1,J) = 'abcd'
C      The string 'abcd' is stored into the character array
C      element S2(1,J)
S2(1,J)(2:3) = 'ef'
C      The string 'ef' is stored into the substring in character
C      positions 2 through 3 of array element S2(1,J) which
C      makes S2(1,J) the string 'aefd'.
```

3.4. Logical Assignment Statement

Purpose:

The logical assignment statement is used to transfer a logical value, that is, either `.TRUE.` or `.FALSE.`, to a logical variable or logical array element.

Form:
$$/ [, /] \dots = / e$$
where:

`/` is a logical variable or logical array element.

`/e` is a logical expression, `character constant` or `Hollerith constant`.

Description:

The result of the logical expression is stored in the target variables appearing to the left of the assignment ("`=`") operator.

No conversions are performed, so the expression must be a logical expression, and the target variables must be logical variables. If `/e` is a `character constant` or `Hollerith constant (a special case)`, then the first word of the constant is stored directly to the target, with no conversion.

The sequence of execution is the same as that for arithmetic assignment statements.

Examples:

```
LOGICAL INDIC, FLAG
INDIC=LEFT .LT. RIGHT
C           The logical result of the logical expression
C           LEFT .LT. RIGHT is stored in the logical variable INDIC.

FLAG=.TRUE.
C           The value .TRUE. is stored in the logical variable FLAG.
```

3.5. Statement Label Assignment (ASSIGN Statement)**Purpose:**

The ASSIGN statement is used to transfer the location of a statement label constant to a variable for subsequent reference in a GO TO statement or an I/O statement.

Form:

```
ASSIGN n TO iv
```

where:

n is an unsigned positive integer indicating an executable or FORMAT statement label.

iv is an unsubscripted integer variable.

Description:

The location of the statement is stored in the target variable that follows TO.

Note that this statement is not the same as an arithmetic assignment. The value of the statement label is not stored and is not subsequently available, such as for output, in that form. Only its location is stored.

The target variable must be of type integer. No conversion is allowed.

The location of the indicated statement label is stored into the target variable in a single operation.

The target variable may be redefined with the same or different statement label value or an integer value.

Example:

```
ASSIGN 10 TO ISTMT
C           The location of statement 10 is stored in variable ISTMT.
```


4. Control Statements

4.1. General

Normal execution of a program unit is sequential, starting with the first executable statement, and continuing with each successive statement until the last executable statement of the program unit.

Control statements change this normal sequence of execution. Some of these statements specify unconditional modifications of program flow while others will change the sequence of execution depending on the results of a test contained within the statement.

FORTRAN control statements are:

- GO TO statements (unconditional GO TO, computed GO TO, assigned GO TO)
- IF statements (arithmetic IF, logical IF)
- Blocking statements (block IF, ELSE IF, ELSE, END IF)
- DO
- CONTINUE
- PAUSE
- STOP
- END
- CALL
- RETURN

CALL and RETURN are not discussed in this section, but are discussed with procedures in Section 7.

Both the control statement and the label referred to must be in the same program unit. A unique statement label may appear on any statement. However, there are rules as to which labels can be referenced (see 10.3.1).

4.2. GO TO Statements

GO TO statements permit transfer of control to an executable statement specified by a statement label cited in the GO TO statement. Control may be transferred either unconditionally or conditionally. Under certain circumstances this transfer can be within the boundaries of a DO statement. This is discussed further in 4.5.5.

The three types of GO TO statements are:

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO

4.2.1. Unconditional GO TO

Purpose:

The unconditional GO TO statement transfers program control to a specified statement.

Form:

GO TO x

where x is the number (statement label) of an executable statement within the same program unit.

Description:

Each execution of an unconditional GO TO statement causes control to be transferred to the statement specified by the statement label.

Any executable statement immediately following this statement should have a statement label. Otherwise, control can never reach such a statement.

Examples:

```
      GO TO 180
110
180   X = X + 1
C           Causes control to be transferred to the statement
C           labeled 180.
      IF (RENT .GT. HIGH) GO TO 100
      OCCUPY = OCCUPY + NEW
100   TOTAL = OCCUPY * RENT
C           If RENT is not greater than the current HIGH,
C           new occupants will be added. The total rent which is
C           collectable is the product of the number of occupants
C           times the cost of renting.
```

```
SUM = 0.  
DO 1 I = 1, 10  
C      This is a list-directed READ  
      READ *, K, J  
      IF (K .EQ. 1) GO TO 14  
      IF (K .EQ. 0) GO TO 15  
      GO TO 1  
14     SUM = SUM + J  
      GO TO 1  
15     SUM = SUM - J  
1     CONTINUE  
      PRINT *, SUM  
      STOP  
      END  
  
C      This sample program uses the unconditional GO TO  
C      statement to either add or subtract from a summation  
C      of 10 numbers.
```

4.2.2. Computed GO TO

Purpose:

A computed GO TO statement transfers control to an indexed member of a statement label list.

Form:

```
GO TO ( x [ , x ] . . . ) [ , ] e
```

or:

```
GO TO ( [ x ] [ , [ x ] ] . . . ) [ , ] e
```

where:

each x is the number (statement label) of an executable statement in the program unit containing the GO TO statement or a variable containing such a number (see 3.5), or is omitted.

e is an integer expression which is used to index a member of the statement label list. The value of e should not exceed the number of x 's appearing in the statement.

Description:

This statement causes control to be transferred to the statement numbered x indexed by e .

x is chosen from the statement label list according to the value of e . If $e = 1$, the first statement label is used; if $e = 2$, the second is used; etc. If e is less than 1 or greater than the number of elements (including empty positions) in the list, or if the selected x is omitted, the execution sequence continues as though a CONTINUE statement were executed.

Examples:

```
INTEGER CHOICE, SALARY, UPCLAS, WEALTH, RICH
GO TO (10,15,20,25), CHOICE
C      Transfer control to statement 10, 15, 20, or 25
C      depending on the value of CHOICE. For example, if the value
C      of CHOICE is 2, control is transferred to statement 15.

C      SALARY is:
C      1 if income is 8,000 or less
C      2 if greater than 8,000 but less than or equal to 12,000
C      3 if greater than 12,000 but less than or equal to 20,000
C      4 if greater than 20,000 but less than or equal to 80,000
C      5 if greater than 80,000
GO TO (100,200,300,400,500) , SALARY
100   LOW = LOW + 1
      .
      .
200   MIDDLE = MIDDLE + 1
      .
      .
300   UPCLAS = UPCLAS + 1
      .
      .
400   WEALTH = WEALTH + 1
      .
      .
500   RICH = RICH + 1
      .
      .
C      Control is transferred to the proper addition depending
C      on the value of SALARY.

SUM = 0
READ *, N, KNODE
DO 1 I = 1, N
  READ *, X
  GO TO (6, 7, 8, 9), KNODE
6    SUM = SUM + X
  GO TO 1
7    SUM = SUM + ALOG(X)
  GO TO 1
8    SUM = SUM + ALOG10(X)
  GO TO 1
9    SUM = SUM + 1. / X
1   CONTINUE
PRINT *, N, KNODE, SUM
STOP
END

C      This program uses the computed GO TO statement to
C      arrange a selection of one of four summation types
C      (sum of the actual values, sum of base e logarithms,
C      sum of base 10 logarithms, and sum of reciprocals).
```

4.2.3. Assigned GO TO

Purpose:

An assigned GO TO statement transfers control to the statement whose label is equal to the current value of a specified variable.

Form:

```
GO TO m [[ , ] ( x[ , x] . . . ) ]
```

where:

x is the statement label of an executable statement in the program unit containing the GO TO statement.

m is a scalar integer variable (not an array element). Its value must be equal to one of the values of *x*, unless the statement label list is omitted.

Description:

At the time of execution of an assigned GO TO statement, the current value of *m* must have been defined to be one of the statements *x* by the previous execution of an ASSIGN statement or by initialization in a DATA or type specification statement. The ASSIGN statement is explained in 3.5. The DATA statement is discussed in 6.8. The value of *m* must identify a statement in the same main program, subroutine, or function as the GO TO statement. No diagnostic will be issued for failure to do so.

Any executable statement immediately following this statement should have a statement label. Otherwise, control can never reach it.

Logically, the assigned GO TO statement can be used whenever a computed GO TO is used (see 4.2.2). The formats differ and the assigned GO TO requires at least one previous ASSIGN statement.

If the statement label list is omitted, the assumed list contains every statement label which has been associated with the variable *m* in an ASSIGN or DATA initialization statement. The result of executing the GO TO statement when *m* does not identify a statement in the list depends on the degree of program optimization being performed, and is not generally predictable.

Examples:

```
INTEGER CHOICE
GO TO CHOICE , (10,15,20,25)
```

C Control is transferred to statement in list whose
C value matches that of CHOICE. For example, if the
C last value assigned to CHOICE was 25, control is
C passed to statement 25.

```
SUM = 0
READ *, N, KNODE
IF (KNODE .EQ. 6) ASSIGN 6 to KSWTCH
IF (KNODE .EQ. 7) ASSIGN 7 to KSWTCH
IF (KNODE .EQ. 8) ASSIGN 8 to KSWTCH
IF (KNODE .EQ. 9) ASSIGN 9 to KSWTCH
DO 1 I = 1, N
  READ *, X
  GO TO KSWTCH, (6, 7, 8, 9)
  GO TO 1
6  SUM = SUM + X
  GO TO 1
7  SUM = SUM + ALOG(X)
  GO TO 1
8  SUM = SUM + ALOG10(X)
  GO TO 1
9  SUM = SUM + 1. / X
1  CONTINUE
PRINT *, N, KNODE, SUM
STOP
END
```

C This sample program uses the assigned GO TO statement
C to arrange a selection from one of four types of
C summation (sum of the actual values, sum of base e
C logarithms, sum of base 10 logarithms, and sum
C of reciprocals).

```
INTEGER GOTO(3), GOTO1, GOTO2, GOTO3
EQUIVALENCE (GOTO(1), GOTO1), (GOTO(2), GOTO2), (GOTO(3), GOTO3)
ASSIGN 10 TO GOTO1
ASSIGN 20 TO GOTO2
ASSIGN 30 TO GOTO3
READ *, I
J = GOTO(I)
GO TO J
```

C This example is incorrect because the statement
C number assigned to J is not associated with J in
C an ASSIGN. The example would be correct
C if the GO TO statement were: GO TO J, (10,20,30).

4.3. IF Statements

IF statements are the decision-making elements in FORTRAN. They test relationships stated within the statement and may modify the normal sequence of execution based on the result of this test.

There are three types of IF statements in FORTRAN: the arithmetic IF, the logical IF, and the block IF. The block IF statement is described along with the rest of the blocking statements in 4.4.

The three types of IF statements have the different forms:

■ Arithmetic IF

$$\text{IF } (a) [x_1] [[x_2] [[x_3]]]$$

where a is any expression of type integer, real, double precision, or typeless.

■ Block IF

$$\text{IF } (e) \text{ THEN}$$

where e is a logical expression.

■ Logical IF

$$\text{IF } (e) \text{ s}$$

where e is a logical expression and s is an executable statement.

4.3.1. Arithmetic IF

Purpose:

The arithmetic IF statement acts as a multidestination branch depending on the condition which is satisfied.

Form:

$$\text{IF } (a) \ x_1, \ x_2, \ x_3$$

or:

$$\text{IF } (a) [x_1] [[x_2] [[x_3]]]$$

where:

a is an arithmetic expression of any type except complex (that is, a is type integer, real, or double precision).

each x is the number (statement label) of an executable statement in the program unit containing the IF statement, or an integer, scalar variable which has been assigned a statement label using the ASSIGN statement (see 3.5). It may also be void (that is, " Δ ").

Description:

The arithmetic IF statement causes transfer of control to the statement numbered x_1 , x_2 , or x_3 when the value of the arithmetic expression (a) is less than, equal to, or greater than zero, respectively.

Any two, or all three, statement labels may be the same. If all three are the same, the statement has the same effect as an unconditional GO TO.

An executable statement immediately following this statement should have a statement label. Otherwise, control can never reach it. Alternately, a void position in the list x_1 , x_2 , x_3 , such as:

```
IF (A) 5,,6
```

implies a reference to the succeeding statement.

Note that if any of the x 's is specified as an integer scalar variable, all three positions must be present (that is, two commas must appear in the list).

Example:

```
SUM = 0
DO 2 I=1,50
  READ *, B
  IF (B) 2,2,1
1    SUM=SUM+B
2    CONTINUE
  PRINT *, SUM
  STOP
END
```

C The basic use of the arithmetic IF statement is to
C discriminate between negative, zero, and positive
C values for variables or expressions.

4.3.2. Logical IF**Purpose:**

The logical IF statement evaluates a logical expression and executes or skips a specified statement depending on whether the value of the expression is true or false, respectively.

Form:

```
IF ( / ) s
```

where:

/ is any logical expression.

s is any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

Description:

Statement *s* is executed if and only if expression *I* is true. If *I* is false, the execution sequence continues as though a CONTINUE statement were executed. Although *s* itself is syntactically a complete statement, it must appear in the same line or set of continuation lines as the clause, IF (*I*).

Examples:

```
      IF (RENT .LE. HIGH) OCCUPY = OCCUPY + 1
      TOTAL = OCCUPY * RENT
C          If RENT is less than or equal to the desired
C          HIGH, the number of new occupants is increased
C          by 1 before the new rent total is computed.
```

```
      SUM = 0
      DO 2 I=1,50
        READ *, B
        IF (B.LE.O.) GO TO 2
        SUM = SUM+B
2      CONTINUE
      PRINT *, SUM
      STOP
      END
```

C The above program sums positive values of B using
C a logical IF statement.

```
      SUM = 0
      DO 2 I=1,50
        READ *, B
2      IF (B.GT.O.) SUM=SUM+B
      PRINT *, SUM
      STOP
      END
```

C This accomplishes the same thing using the greater than
C (.GT.) operator.

4.4. Blocking Statements

The block IF, ELSE IF, ELSE, and END IF statements define an IF-THEN-ELSE blocking structure. These blocking statements allow the FORTRAN programmer to conditionally execute a block of statements. With the proper use of this feature, the GO TO statement will seldom be necessary. This will allow more structure in FORTRAN programs, thus making them more reliable and understandable.

Indentation of lines in the source code will make programs with blocking statements more readable (see examples in 4.4.5).

A block begins with a block IF statement and ends with an END IF statement. Between the two statements may appear zero or more ELSE IF statements and zero or one ELSE statements. The ELSE statement (if it appears) must follow all ELSE IF statements (if any) in the block. Executable statements may appear in the blocks between the blocking statements. Blocks may be nested.

Subsection 4.4.5 contains examples which show the FORTRAN IF-THEN-ELSE blocking statements. The examples use all of the blocking statements (block IF, ELSE IF, ELSE, and END IF), and also show the different types of blocks (IF-block, ELSE IF-block, ELSE-block) and IF-level nesting.

4.4.1. Block IF Statement

Purpose:

The block IF statement is used with the END IF statement and, optionally, the ELSE IF and ELSE statements, to control the execution sequence.

Form:

IF (*e*) THEN

where *e* is a logical expression.

4.4.1.1. IF-Level

The IF-level of a statement *s* is:

$$n_1 - n_2$$

where n_1 is the number of block IF statements from the beginning of the program unit up to and including *s*, and n_2 is the number of END IF statements in the program unit up to but not including *s*.

The IF-level of every statement must be zero or positive. The IF-level of each block IF, ELSE IF, ELSE, and END IF statement must be positive. The IF-level of the END statement of each program unit (or the last statement in the program unit, unless the last statement is END IF) must be zero.

The maximum IF-level allowed at any point in a program unit is 25.

4.4.1.2. IF-Block

An IF-block consists of all the executable statements after the block IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement. An IF-block may be empty.

4.4.1.3. Execution of a Block IF Statement

Execution of a block IF statement causes evaluation of the expression *e*. If the value of *e* is true, normal execution continues with the first statement of the IF-block. If the value of *e* 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 *e* 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.

Transfer of control into an IF-block should not be made from outside the IF-block.

If the execution of the last statement in the IF-block does not result in a transfer of control, then control is transferred to the next END IF statement that has the same IF-level as the block IF statement that immediately precedes the IF-block.

4.4.2. ELSE IF Statement

Form:

ELSE IF (*e*) THEN

where *e* is a logical expression.

4.4.2.1. ELSE IF-Block

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.

4.4.2.2. Execution of an ELSE IF Statement

Execution of an ELSE IF statement causes evaluation of the expression *e*. If the value of *e* is true, normal execution sequence continues with the first statement of the ELSE IF-block. If the value of *e* 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 *e* 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 should not be made from outside the ELSE IF-block. The statement label, if any, of the ELSE IF statement must not be referred to by any statement **except DELETE**.

If execution of the last statement in the ELSE IF-block does not result in a transfer of control, then control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement that immediately precedes the ELSE IF-block.

4.4.3. ELSE Statement

Form:

ELSE

4.4.3.1. ELSE-Block

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.

An END IF statement of the same IF-level as the ELSE statement must appear before the appearance of an ELSE IF or ELSE statement of the same IF-level.

4.4.3.2. Execution of an ELSE Statement

Execution of an ELSE statement has no effect. Normal execution sequence continues.

Transfer of control into an ELSE-block should not be made from outside the ELSE-block. The statement label, if any, of an ELSE statement must not be referred to by any statement, **except DELETE.**

4.4.4. END IF Statement

Form:

END IF

Description:

Execution of an END IF statement has no effect. Normal execution sequence continues.

For each block IF statement, there must be a corresponding END IF statement in the same program unit. A corresponding END IF statement is the next END IF statement that has the same IF-level as the block IF statement.

4.4.5. Examples Using the Blocking Statements

Example 1:

C - In this example, if A and B are equal, then
C - the three statements in the IF-block (that is, the
C - statements between the block IF and END IF statements)
C - are executed, and then the statement after the END IF is
C - executed. If A and B are not equal, then the
C - statement after the END IF is executed.
C

```
IF (A .EQ. B) THEN
    I = I+1
    A = B+C
    CALL SUB1 (I,A)
END IF
```

The following three sequences of statements (Examples 2 through 4) are equivalent. L1 and L2 are logical scalar variables in the examples which follow. The examples cause I and J to be set depending on the values of L1 and L2. If L1 is .TRUE., then I and J are set to 1 and 2, respectively. If L1 is .FALSE. and L2 is .TRUE., then I and J are set to 2 and 3, respectively. If L1 and L2 are both .FALSE., then I and J are set to 3 and 4, respectively.

Example 2:

- C - This example uses all of the
- C - blocking statements:
- C - BLOCK IF, ELSE IF, ELSE, and END IF.
- C - The maximum IF-level is one.

C

```
IF (L1) THEN
    I = 1
    J = 2
} IF-block
ELSE IF (L2) THEN
    I = 2
    J = 3
} ELSE IF-block
ELSE
    I = 3
    J = 4
} ELSE-block
END IF
```

Example 3:

- C - This example uses all of the
- C - blocking statements except ELSE IF.
- C - The maximum IF-level is two.

C

```
IF (L1) THEN
    I = 1
    J = 2
ELSE
    IF (L2) THEN
        I = 2
        J = 3
    ELSE
        I = 3
        J = 4
    END IF
END IF
```

Example 4:

- C - This example uses none of the
- C - blocking statements

C

```
IF (.NOT. L1) GO TO 10
    I = 1
    J = 2
    GO TO 30
10 IF (.NOT. L2) GO TO 20
    I = 2
    J = 3
    GO TO 30
20 I = 3
    J = 4
30 CONTINUE
```

4.5. DO Statement

Purpose:

A DO statement is used to specify a loop, called a DO-loop. This loop controls repeated execution of a set of executable statements.

Form:

DO *s* [,] *i* = *e*₁, *e*₂ [, *e*₃]

where:

s is the statement label of an executable or **FORMAT** statement. The statement identified by *s*, called the terminal statement of the DO-loop, shall follow the DO statement in the sequence of statements within the same program unit as the DO statement. The terminal statement of a DO-loop shall not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement.

i is the name of an integer, real, or double precision variable, called the DO-variable.

*e*₁ is an expression indicating the initial value for the DO-variable *i*.

*e*₂ is an expression indicating the terminal test value for the DO-variable *i*.

*e*₃ is an expression indicating the increment value for the DO-variable *i*. *e*₃ must not be zero. If *e*₃ is omitted, it is assumed to have a value of one.

*e*₁, *e*₂, and *e*₃ are each an integer, real, or double precision expression.

4.5.1. Range of a DO-Loop

The range of a DO-loop consists of all of the executable statements that appear following the DO statement that specifies the DO-loop, up to and including the terminal statement of the DO-loop. **If the terminal statement is nonexecutable (that is, FORMAT), the range of the DO is not extended to include the next executable statement.**

If a DO statement appears within an IF-block, ELSE IF-block, or ELSE-block, the range of that DO-loop shall be contained entirely within that IF-block, ELSE IF-block, or ELSE-block, respectively.

If a block IF statement appears within the range of a DO-loop, the corresponding END IF statement shall also appear within the range of that DO-loop.

4.5.2. Nested DO-Loops

DO statements are permitted within the range of DO-loops. Whenever a DO statement is placed within the range of a DO-loop, the following rules must be observed:

- If a DO-loop contains another DO-loop, the range of the second DO-loop must be entirely within the range of the first.
- The range of an inner DO-loop may, however, contain the last statement in the range of the next outer DO-loop. Note that when DO-loops are nested in this manner only iterations of the inner DO-loop may be terminated by a transfer of control to the last statement of the loop.

Such a set of DO-loops is called a DO-nest. DO-loops and implied DO-loops may be nested to a maximum depth of 25 loops.

A nest of DO-loops is considered completely nested if no loop in the nest terminates before the last loop begins. For example, the following two nests are completely nested:

```

      DO 50 I = 1, 4
        A (I) = B (I)**2
        DO 50 J=1, 5
          C(I,J) = A (I)
        DO 10 I = L, M
          N = I + K
          DO 15 J = 1, 100, 2
            TABLE(J,I) = SUM(J,N)-1
          B(N) = A(N)

```

} Range of inner DO } Range of outer DO
 } Range of inner loop } Range of outer DO

The following DO-loops are not completely nested:

```

      DO 100, I = 1, 10
        DO 200, J = 2, 12
          .
          .
          .
        CONTINUE
        DO 300, K = 1, 10
          .
          .
          .
        CONTINUE
      CONTINUE

```

} inner loop } outer DO
 } inner loop }

The following is not properly nested, and is thus in error:

```

      DO 100, I = 1, 10
        M = I + N
        DO 200, J = 10, 1, -1
          INDEX(I,J) = TABLE(M,J)
          A(J) = Z(N-J)

```

} Range of first DO }
 } Range of second DO }

4.5.3. Active and Inactive DO-Loops

A DO-loop is either active or inactive. Initially inactive, a DO-loop becomes active only when its DO statement is executed.

Once active, the DO-loop becomes inactive only when:

- its iteration count is tested (4.5.4.2) and determined to be zero;
- a RETURN statement is executed within its range;
- control is transferred to a statement that is in the same program unit and is outside the range of the DO-loop; or
- any STOP statement in the executable program is executed, or execution is terminated for any other reason (for example, error or end-of-file conditions on I/O statements).

Execution of a function reference or CALL statement that appears in the range of a DO-loop does not cause the DO-loop to become inactive, except when control is returned by means of an alternate return specifier (that is, RETURN *i*) to a statement that is not in the range of the DO-loop.

When a DO-loop becomes inactive, the DO-variable of the DO-loop retains its last defined value.

4.5.4. DO-Loop Execution

Execution of a DO-loop involves the following steps:

- executing the DO statement
- loop control processing
- execution of the range
- terminal statement execution
- incrementation processing

These steps are described in the paragraphs which follow.

4.5.4.1. Executing a DO Statement

The effect of executing a DO statement is to perform the following steps in sequence:

1. The initial parameter m_1 , the terminal parameter m_2 , and the incrementation parameter m_3 are established by evaluating e_1 , e_2 , and e_3 , respectively, including, if necessary, conversion to the type of the DO-variable according to the rules for arithmetic conversion. Note that if e_3 is not specified, m_3 is assumed to be one.
2. The DO-variable becomes defined with the value of the initial parameter m_1 .
3. The iteration count is established and is the value of the expression:

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

Note that the iteration count is zero (that is, the DO-loop range will be executed zero times) whenever:

$$m_1 > m_2 \text{ and } m_3 > 0$$

or:

$$m_1 < m_2 \text{ and } m_3 < 0.$$

Note that since the iteration count is calculated only once (before entry into the loop), variables appearing in the expressions e_1 , e_2 , and e_3 may be changed during execution of the loop with no effect on the iteration count.

The user is cautioned that if noninteger expressions for e_1 , e_2 , and e_3 are used, the iteration count may be different than expected due to the approximate nature of reals and the integer conversion operation.

At the completion of execution of the DO statement, loop control processing begins.

4.5.4.2. Loop Control Processing

Loop control processing determines if further execution of the range of the DO-loop is required. The iteration count is tested. If it is not zero, execution of the first statement in the range of the DO-loop begins.

If the iteration count is zero, the DO-loop becomes inactive. If, as a result, all of the DO-loops sharing the terminal statement of this DO-loop are inactive, normal execution continues with execution of the next executable statement following the terminal statement. However, if some of the DO-loops sharing the terminal statement are active, execution continues with incrementation processing (see 4.5.4.5).

4.5.4.3. Execution of the Range

Statements in the range of a DO-loop are executed until the terminal statement is reached. Except by incrementation (see 4.5.4.5), the DO-variable of the DO-loop may neither be redefined nor become undefined during execution of the range of the DO-loop. It is the user's responsibility to ensure that such redefinitions do not occur.

4.5.4.4. Terminal Statement Execution

Execution of the terminal statement occurs as a result of the normal execution sequence or as a result of transfer of control. Unless execution of the terminal statement results in a transfer of control, execution then continues with incrementation processing.

4.5.4.5. Incrementation Processing

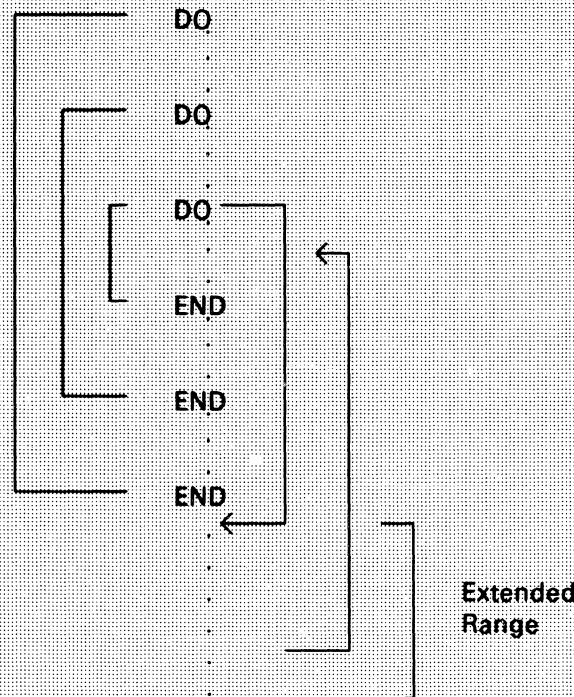
Incrementation processing has the effect of the following steps performed in sequence:

1. The DO-variable, the iteration count, and the incrementation parameter of the active DO-loop whose DO statement was most recently executed, are selected for processing.
2. The value of the DO-variable is increased by the value of the incrementation parameter m_3 .
3. The iteration count is decreased by one.
4. Execution continues with loop control processing (see 4.5.4.2) of the same DO-loop whose iteration count was decreased.

4.5.5. Extended Range of a DO-Loop

The extended range of a DO-loop is defined as those statements that are executed between the transfer out of the innermost DO-loop of a set of completely nested DO-loops and the transfer back into the range of this innermost DO-loop. In a set of completely nested DO-loops, the first DO-loop is not in the range of any other DO-loop, and each succeeding DO-loop is in the range of every DO-loop which precedes it. The following restrictions apply:

- Transfer into the range of a DO-loop is permitted only if such a transfer is from the extended range of the DO-loop.
- The extended range of a DO-loop must not contain another DO statement that has an extended range, if the second DO statement is within the same program unit as the first.
- The DO-variable cannot be changed in the extended range of the DO.
- Transfer into extended range must be from the innermost loop. See the following illustration:



- A statement that is the end of the range of more than one DO-loop is within the innermost DO. The statement label of such a terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.
- The use of, and return from, a subprogram (see 7.1) from within any DO-loop in a nest of DOs, or from within an extended range, is permitted. However, if the DO-variable of the loop is passed to the subprogram, the subprogram should not alter its value.

NOTE: *The extended range of a DO-loop is not considered good programming practice and should not be used in new programs.*

4.5.6. Availability of the DO-Variable Value

If optimization is not selected, the memory location associated with the DO-variable of a DO-loop will always contain the current value of the DO-variable.

If optimization is selected, the DO-variable of a DO-loop (as well as other variables) may be maintained in a machine register. Consequently, the storage location associated with the variable may not always contain the current value of the variable. This will not be noticeable to the user unless the storage location associated with the variable is dumped. The variable will be in storage if a reference requiring the variable to be used from storage has been performed.

The following types of references require that the variable be used from storage.

- The variable appears in an input/output list within the loop other than as part of a subscript.
- The variable is used as an argument of a subprogram referenced within the loop.
- The variable is used outside the loop before being redefined and there is a branch to a statement outside the range of the DO-loop.

4.5.7. DO-Loop Examples

```
DO 100 I = 1, 50
100  CURRENT(I)=CURRENT(I) - OUT(I)
    X=Y*Z
```

C In this example, execution of the DO statement causes the
C initialization of (1) the DO-variable I to 1, and (2) the
C iteration count to 50. After statement 100 is
C executed each time through the loop, incrementation
C processing causes (1) I to be incremented by 1, and
C (2) the iteration count to be reduced by 1. After
C 50 executions of the DO-loop, incrementation processing
C causes (1) I to be set to 51, and (2) the iteration
C count to be set to 0. Loop control processing then
C causes the DO-loop to become inactive, thus causing
C execution of the third statement. I contains the value 51.

```
DO 200, J = 10, 1, -2
    I = J + K
200  ROW (I) = COL(I)
    X=Y*Z
```

C In this example, execution of the DO statement causes the
C initialization of (1) the DO-variable J to 10, and (2) the
C iteration count to 5. Incrementation processing
C (after statement 200) causes (1) J to be incremented by
C -2, and (2) the iteration count to be reduced by 1.
C After five executions of the DO-loop, incrementation
C processing causes (1) J to be set to 0, and (2) the
C iteration count to be set to 0. Loop control processing
C then causes the DO-loop to be inactive, thus causing
C execution of the fourth statement. J contains the value 0.

```
N=0
DO 100 I=1,10
  J=1
  DO 100 K=1,5
    L=K
100    N=N+1
101  CONTINUE
C      After execution of these statements and at the execution of
C      the CONTINUE statement, I=11, J=10, K=6, L=5, and N=50.

N=0
DO 200 I=1,10
  J=1
  DO 200 K=5,1
    L=K
200    N=N+1
201  CONTINUE
C      After execution of these statements and at the execution of
C      the CONTINUE statement, I=11, J=10, K=5, and N=0. The value
C      of L is not changed by these statements, since the
C      iteration count for
C      the innermost loop is 0
C      (that is, the innermost loop is traversed zero times,
C      since the increment value is not specified and
C      is assumed to be one. See 4.5.4.1).
C
C      This example demonstrates some of the effects of real-valued
C      DO loops and subscripts.
C      Note the effects of using a rounding factor.

DIMENSION I(12)
DATA I/1,2,3,4,5,6,7,8,9,10,11,12/
A = 2.4
B = 4.0
C = 3.0
DO 100 R = A/2.0, B*C, 1.2
100  PRINT *, I(R), I(R+.0001), R
DO 200 R = A/2.0, B*C+.0001, 1.2
200  PRINT *, I(R), I(R+.0001), R
END
```

@ rounding factor

Without Rounding Factor	With Rounding Factor	R	
1	1	1.2000000	} 9 iterations
2	2	2.4000000	
3	3	3.6000000	
4	4	4.8000000	
5	6	5.9999999	
7	7	7.1999999	
8	8	8.3999999	
9	9	9.5999998	
10	10	10.8000000	
1	1	1.2000000	} 10 iterations (rounding factor added to terminal parameter)
2	2	2.4000000	
3	3	3.6000000	
4	4	4.8000000	
5	6	5.9999999	
7	7	7.1999999	
8	8	8.3999999	
9	9	9.5999998	
10	10	10.8000000	
11	12	12.0000000	

4.6. CONTINUE Statement

Purpose:

The CONTINUE statement acts as a dummy executable statement.

Form:

CONTINUE

Description:

The CONTINUE statement does not perform any executable function.

This statement may be placed anywhere in the source program where an executable statement may appear. It does not affect the sequence of program execution.

A statement label is usually used with the CONTINUE statement. In this manner, it provides a point to which control can be transferred without implying any executable action.

CONTINUE is primarily used as the terminal statement of a DO-loop range. A transfer of control from any point within the loop to the CONTINUE statement allows the completion of an iteration of the loop without specifying an additional action to be taken. Use of the CONTINUE statement in this way can facilitate program updating.

Example:

```
16      DO 29 I=1,10
```

```
          IF (A) 29,38,34
29      CONTINUE
```

```
C          This example illustrates the use of CONTINUE as the
C          terminal statement for a DO-loop.
```

4.7. PAUSE Statement**Purpose:**

The PAUSE statement temporarily suspends execution of a demand (that is, interactive) program.

Form:

```
PAUSE [ { n } ]
       [ { message } ]
```

where:

n is a string of decimal digits. FORTRAN 77 allows a string of one to five decimal digits. ASCII FORTRAN allows a string of one to six decimal digits.

message is a literal constant enclosed in apostrophes and containing alphanumeric or special characters. Within the literal, an apostrophe can be indicated by two successive apostrophes. The maximum length of *message* is 124 characters.

Description:

A PAUSE of either form temporarily halts execution of the program.

The program waits until the user transmits a carriage return which causes the program to resume execution, starting with the next statement after the PAUSE statement.

PAUSE *n*, PAUSE *message*, or PAUSE 00000 is displayed upon the demand terminal, depending upon whether *n*, *message*, or no parameter was specified, respectively. Program execution is suspended until input is received from the terminal.

If the program is not in demand mode, the PAUSE *n*, the PAUSE *message*, or PAUSE 00000 will be displayed on the system print file, and program execution will continue.

Example:

```
60     IF (A) 80,90,110
70     CONTINUE
80     STOP 'A IS NEGATIVE'
90     PAUSE 'A IS 0'
100    GO TO 180
110    A=B**2+C**3
120    GO TO 60
```

```
C           This example will cause execution to pause at statement 90
C           if A=0. When the user causes the program to resume
C           execution, the next statement executed will be the
C           statement at statement label 100.
```

4.8. STOP Statement**Purpose:**

STOP terminates the execution of the program.

Form:

```
STOP [ { n
       message } ]
```

where:

n is a string of decimal digits. FORTRAN 77 allows a string of one to five decimal digits. ASCII FORTRAN allows a string of one to six decimal digits.

message is a literal constant enclosed in apostrophes and containing alphanumeric or special characters. Within the literal, an apostrophe is indicated by two successive apostrophes. The maximum length of *message* is 124 characters.

Description:

Execution of the STOP statement will terminate the execution of the program.

If *n* or *message* is specified, STOP *n* or STOP *message* will be displayed on the system print file. Otherwise, nothing will be displayed.

Note that all open files are closed as part of program termination.

Example:

```
60     IF (A) 80,90,110
70     CONTINUE
80     STOP 'A IS NEGATIVE'
90     PAUSE 'A IS 0'
100    GO TO 180
110    A=B**2-C**3
120    GO TO 60
```

```
C           Execution of this program will cease at statement label 80
C           if A is negative.
```

4.9. END Statement

Purpose:

The END statement indicates the end of a program unit. In ASCII FORTRAN, the END statement indicates the end of compilation for a particular group of program units.

Form:

END

Description:

The END statement terminates a group of program units, consisting of an external program unit and zero or more internal subprograms. An END statement must appear at the end of:

- an external program unit (main program, or external function or subroutine) that has no associated internal subprograms
- the last internal subprogram associated with a given external program unit
- a BLOCK DATA program

If present, the END statement must physically be the last statement in a program unit. It indicates to the compiler that the preceding statements are to be compiled as one group of program units. It also indicates that the following program unit (if any) is an external program unit.

The execution of an END statement implies a RETURN in a subprogram or a STOP in a main program.

If the source input to the compiler contains more than one program unit, then each external program unit must be terminated by an END statement, a FUNCTION statement, or a SUBROUTINE statement. The FUNCTION and SUBROUTINE statements in this case signal the start of an internal subprogram as well as the end of the previous program unit (see 7.4.2 and 7.4.3).

If the END statement is missing, a control card signals the physical end of the source input to the compiler.

5. Input/Output Statements

5.1. General

Input statements obtain data for program use from input files. Output statements store results produced in the program in output files. These files may be information storage files within the computer system (internal storage) or they may be devices such as keyboards, printers, display terminals, or other peripheral devices. Therefore, input/output statements can be used to transfer data:

- from internal storage to an output device,
- from an input device to internal storage, or
- from internal storage to internal storage.

Data on an input/output device composes a file. Files are composed of one or more records. A sequential input/output device may contain one or more files.

Depending on the file being manipulated, an input/output statement is one of the three types:

- Sequential
- Direct
- Internal

The sequential input/output statements are READ, WRITE, PRINT, PUNCH, BACKSPACE, ENDFILE, REWIND, OPEN, CLOSE, INQUIRE, and DEFINE FILE. They are used for accessing files sequentially. Using sequential input/output statements, it is not possible to read, for example, the seventh record of a file directly; it is necessary to indicate that the preceding six records have been passed over. Once the seventh record is read, it may be impossible (as in the case of a card reader) to go back and reread the fourth record. However, this can be done where the BACKSPACE or REWIND statement is effective (as on magnetic tape).

Direct input/output statements are READ, WRITE, FIND, OPEN, CLOSE, INQUIRE, and DEFINE FILE. They are used for referring to random access files in any order based on the record number.

The internal input/output statements are READ, WRITE, ENCODE, and DECODE. They are used for internal storage-to-internal storage transfers.

Four types of records may be read or written depending on the I/O statement type used:

<u>Record Types</u>	<u>I/O Statement Types Allowed</u>
Formatted	Sequential, direct, internal
Unformatted	Sequential, direct
Namelist	Sequential
List-directed	Sequential, internal (ENCODE and DECODE only)

The type of record that is to be read or written is determined from the form of the READ or WRITE statement.

Formatted records are read or written under the control of a FORMAT statement which describes the characteristics of the data being transferred. On output, data is transformed from machine representation to coded form. On input, data is transformed from coded form to machine representation. See 5.3 for a description of the FORMAT statement.

Unformatted records are read or written without format control. The data is read or written in machine representation form. No transformation is done on input or output.

Namelist records are read or written under the control of the NAMELIST statement. See 5.4 for details of the NAMELIST statement.

List-directed records are read or written with an implied format control. No FORMAT statement is required. See 5.5 for a description of list-directed input/output.

Each execution of an input/output statement causes a new record to be processed.

All character data is assumed to be in the ASCII character code.

The default record sizes are:

APRINT-APRNTA-AREADA symbionts	132	characters
AREAD symbionts	80	characters
APUNCH-APNCHA symbionts	80	characters
System Data Format (SDF) files	33	words (132 characters)
ANSI files	132	characters

If the default record sizes are to be exceeded, an OPEN or DEFINE FILE statement must be used.

In addition to the information in this section, more detailed information on input/output is contained in Appendix G.

5.2. Elements of Input/Output Statements

Input/output statements are composed of FORTRAN key words (READ, WRITE, etc.) and a control list which contains control specifications. The control specifications allowed on an input/output statement are dependent on the type of input/output statement being used, the type of file access, and the type of record desired. The various control specifications are:

- file reference number specification
- record number specification
- input/output list specification
- format specification
- **namelist name specification**
- ERR clause specification
- END clause specification
- IOSTAT clause specification

The complete input/output statements are described in 5.6, 5.7, 5.9, and 5.10.

5.2.1. File Reference Number Specification

The FORTRAN programmer refers to a file by its file reference number. The file reference number is specified in the input/output statement to indicate to which file the statement refers. The form of a file reference number specification is:

[UNIT =] *u*

The UNIT= clause is optional. If that clause is missing, the unit number *u* must be the first item in a list of specifiers.

The file reference number for an external file may be specified as an unsigned integer constant or an integer expression whose value shall be greater than or equal to zero. An asterisk may be used to designate the symbiont units 5 and 6 (the symbionts AREAD\$ and APRINT\$, see G.6) when doing formatted sequential READ and WRITE statements. The asterisk may not be used for the unit number in auxiliary input/output statements.

The file reference for an internal file may be a character variable, character array, character array element, or character substring.

The file reference number is sometimes referred to as the file reference number.

Once a file reference number is associated with a file and that file is opened, the file reference number may not be associated with another file until the first file is closed via either the CLOSE statement or the CLOSE service subroutine.

There is no standard convention for numbering of system files. A particular site may establish its own convention for assignment of file numbers to input/output media (that is, card readers, printers, etc.). However, unless changed by the site, the default assignment of standard reference units is:

<u>Number</u>	<u>Unit</u>
5	Standard Input (AREAD\$)
6	Standard Print (APRINT\$)
1	Standard Punch (APUNCH\$)
0	Reread

Information on other units and the assignment of reference numbers is contained in Appendix G.

Note that the external file may not be word-addressable mass storage.

5.2.2. Record Number Specification

The direct access input/output statements READ, WRITE, and FIND require the programmer to specify the relative position (index) of the record in the file. The relative position of the record is specified as an integer expression.

The forms of the record number specification are:

REC = *rn*

or:

rn

If the second form of the record number specification is used, the UNIT= optional clause must not appear before the file reference specification.

5.2.3. Input/Output List Specifications

An input/output list is composed of list items separated by commas. A list item may be a variable name, an array name, an array element, a character substring name, an expression (output list only), or an implied-DO list. The name of an assumed size dummy array must not appear as an input/output list item. If a function reference is used in an output list, the function and any subprogram which it invokes must not perform input/output. An attempt to perform recursive I/O will result in a fatal error message. A character expression involving concatenation of an operand whose length specification is an asterisk in parentheses is not allowed in an output list according to the FORTRAN 77 standard, although ASCII FORTRAN allows it.

An output list determines which variables (storage locations) are to be written to the output record when a WRITE statement is executed. An input list determines which variables (storage locations) are to be filled from the input record when a READ statement is executed. The positioning of the names in the input/output list specifies the order in which the data will be transferred between the record and the variables (storage locations).

If a variable name or array element appears in the list, one item is transmitted between the storage location and a record. If an array name appears in the list, the entire array is transmitted in the order in which it is stored (column-major order). If the array has more than one dimension, it is stored in ascending storage locations with the value of the first subscript increasing most rapidly and the value of the last subscript increasing least rapidly. This ordering is described in 2.2.2.4.5.

Parts of arrays can be read or written using an implied-DO clause in the input/output list specification.

On output, if numeric values fail to fit in the specified output field, the field will be filled with asterisks and no error or warning message will be given.

The implied-DO clause enables selected array elements to be referenced for input/output operations without putting each value on a separate record (for example, if the statement were in a DO-loop) or listing each element individually.

An input/output list specification containing the implied-DO clause has the form:

(input/output list , $i = e_1 , e_2 [, e_3]$)

The elements i , e_1 , e_2 , and e_3 are as specified for the DO statement (see 4.5). The range of an implied-DO specification is the input/output list of the implied-DO. For input lists, i or elements of e_1 , e_2 , and e_3 may not appear as input list items within the range of the implied-DO.

An example of an input/output list without an implied-DO clause is:

VAR1, ARRAY1, ARRAY2(5)

This list refers to input/output of the contents of variable VAR1, array ARRAY1 and element five of array ARRAY2.

The list:

(ARRAY2(J), J = 1,3)

refers to the first, second and third elements of ARRAY2 without having to specify each element separately.

The list:

((ARRAY4(J,K), J = 1,9,4), K = 2,3)

refers to elements ARRAY4(1,2), ARRAY4(5,2), ARRAY4(9,2), ARRAY4(1,3), ARRAY4(5,3), and ARRAY4(9,3) in that order.

The list:

VAR1, VAR2, (VAR3, ARRAY5(J), J = 3,6)

refers sequentially to VAR1, VAR2, VAR3, ARRAY5(3), VAR3, ARRAY5(4), VAR3, ARRAY5(5), VAR3, and ARRAY5(6).

5.2.4. Format Statement Specification

In order to read or write formatted records, one of the following format specifiers must be present in the READ or WRITE statement:

- Statement label of a FORMAT statement
- Name of an integer variable containing the statement label (assigned with the ASSIGN statement) of a FORMAT statement

- Name of a character array (except assumed-size array) or scalar character variable containing a format specification
- Name of a noncharacter array (except assumed-size array) containing a format specification
- Character expression containing the format
- Asterisk specifying list-directed formatting

The form of the format statement specifier is:

[FMT =] *f*

If the optional FMT= clause is omitted from the format statement specifier, the format specifier *f* must be the second item in the list of specifiers, and the unit specifier must be the first item without the optional UNIT= clause. If the UNIT= and FMT= clauses are present, the specifiers in the control list may be in any order.

The format specifications are described in 5.3. If the format specifier identifies a FORMAT statement, it must be in the same program unit as the input/output statement.

5.2.5. Namelist Name Specification

To read or write namelist records, it is necessary to specify a NAMELIST statement name in the READ or WRITE statement.

A namelist name is a symbolic name (see 2.2.2). By referring to this name, it is possible to write a simple input/output statement which has the same effect as an input/output statement with a long list and a reference to a complicated FORMAT statement. The NAMELIST statement is described in 5.4.

5.2.6. ERR Clause Specification

Certain input/output statements allow the programmer to specify an error clause. The error clause specifies a statement label (in the same program unit as the input/output statement) to which transfer is made if an error or warning occurs during execution of the input/output statement. If an error occurs while executing an input/output statement and neither an error clause nor an IOSTAT clause is specified, the program will be terminated.

The error clause has the form:

ERR = *s*

where *s* is the statement label to which control is to be transferred when an error or warning condition is detected.

5.2.7. END Clause Specification

An END clause is allowed in certain input/output statements. The END clause specifies a statement label (in the same program unit as the input/output statement) to which transfer is made if an end-of-file condition is encountered during execution of the input/output statement. If an end-of-file condition is encountered while executing an input/output statement and neither an END clause nor an IOSTAT clause is specified, the program will be terminated.

The END clause has the form:

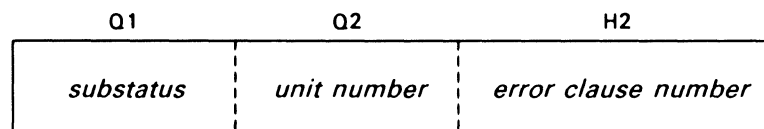
END = *sn*

where *sn* is the statement label to which control is to be transferred.

5.2.8. Input/Output Status Clause Specification

An input/output status specifier is an integer variable or integer array element which, if specified in an input/output statement, will receive a value determined by the success of the execution of the statement. The value returned will be one of the following:

- a zero will be returned if neither an error condition nor an end-of-file condition is encountered.
- the value of the I/O status word, PTIOE, from the storage control table will be returned if an error or warning condition is encountered. The value returned has the format:



See G.9 for a further discussion of PTIOE.

- a negative one is returned if no error condition is encountered but an end-of-file condition is encountered.

The input/output status clause has the form:

IOSTAT = *ios*

where *ios* is an integer variable or integer array element.

If the IOSTAT clause is present, control will be returned to the user after an error or warning condition is encountered regardless of the presence or absence of an ERR or END clause. Control will return inline if only the IOSTAT clause is present.

5.3. FORMAT Statement

Purpose:

The FORMAT statement specifies the external form of the values of the input/output list elements and the arrangement of the data within the records to be transmitted.

Form:

```
f FORMAT ( [ format-specifications ] )
```

where *f* is a statement label (required); *format-specifications* describe the form of the data. These are detailed in 5.3.1. If the format specifications are omitted (that is, FORMAT()) and the input/output list is not empty, the input or output is list-directed (see 5.5). If no input/output list is present and the format specification is empty, one input record is skipped or one blank output record is written.

Description:

This is a nonexecutable statement that may be placed anywhere in the source program. Since statement labels are local to the routine in which they appear, FORMAT statements are local to the routine by implication.

Format specifications describe fields to be input or output. A field is a string of adjacent character positions. The width of a field is the number of character positions in the string. A formatted record is a character string. On output, a record is constructed, essentially, by concatenating the output fields. On input, the record is divided into fields according to the format specification. A field may contain the character representation of the value of a simple list item, it may contain literal characters, or it may be skipped (filled with the character blank).

For any field, the format specification must define its width and the type of conversion between the internal and the external forms, or the literal characters desired, or it must indicate that the field is to be skipped.

Format specifications may include the following:

- Editing codes and repetition factors
- Sign options
- Grouping delimiters
- Scale factors
- Carriage controls
- Line delimiters.

FORMAT statements are examined by the compiler for correctness and converted to a more efficient internal form which is used during the execution of the FORTRAN program.

As stated in 5.2.4, a character array, scalar character variable name, character expression, or noncharacter array may be used in place of a statement label of a FORMAT statement in a formatted input/output statement. The contents are format specifications which may be modified during the execution of the object program. Such formats are not converted to an internal form by the compiler and are not examined by the compiler for correctness. The rules governing the contents of such formats are presented in 5.3.9.

Examples:

See 5.3.1.

5.3.1. Editing Codes

An editing code is used to specify the characteristics of a field. It can specify the type of conversion and field width, the explanatory literal characters, or the number of characters to be skipped. If two or more editing codes are used in a format, they must be separated from each other by a comma, a colon, or a slash except as noted in the following. The size of the integer constants w , d , p , and e must be less than 512. The editing codes are:

<u>Code</u>	<u>Usage</u>
Iw	Integer. The field is to occupy w positions, the type of list item is integer, and the value of the list item is to appear as an integer constant right justified in the field. If the field width specified for output is not large enough to contain the entire integer including a sign if it is required, the field is asterisk filled to indicate overflow. If the field width specified is larger than that required for the constant, it is blank-filled on the left. In an input field, leading blanks are ignored while embedded blanks are interpreted as zeros unless the BN format has been encountered or BLANK=NULL occurred on the OPEN statement (see 5.10.1).
$Iw.d$	Integer. This code is ignored on input. The w is the same as for Iw . The d indicates that at least d digits will be written. This will include any zeros needed to equal d digits. The difference between w and d will be space filled on the left of the field. If a sign is required, w must be greater than d .
Jw	Integer zero-filled. This code is the same as Iw for input. For output, the integer is right-adjusted as in an Iw field; however, the rest of the field is zero-filled. If there is a sign, it appears in the leftmost position in the field.
$Fw.d$	External fixed-point. The field is to occupy w positions on output. The list item is real, one part of a complex value, or double precision. The integer portion of the corresponding value appears as a right-adjusted real constant in the leftmost $w-d-1$ positions in the field. This integer part may not have an exponent. A decimal point character occupies position $d+1$ from the right-hand end of the field ($d \geq 0$). The fractional part of the number is to occupy d digits written to the right of the decimal point. If the number is negative, the field width must permit a minus sign to be written immediately to the left of the most significant digit of the number. If the field width specified is larger than that necessary to contain the number, it is space filled on the left. If the width specified is not large enough to contain the number including any necessary sign, the field is asterisk-filled. If a sign is required, the minimum field width necessary for an F editing code is $w = 2 + d + s$ where $s = \text{MAX}(p + i, 0)$, p is the scale factor, and i is defined by $10^{i-1} \leq M < 10^i$ where M is the quantity to be written.

On input, $Fw.d$ designates a field of w characters, of which d characters follow the assumed decimal point. The assumed decimal point is ignored if a decimal point appears in the input. An exponent may follow the number. It must be a signed integer constant or begin with an E or D followed by an optionally signed integer constant. Blanks embedded in the input field are treated as zeros unless the BN format has occurred or the OPEN statement for the file contained a BLANK=NULL. Embedded and trailing blanks will then be ignored. Leading blanks are ignored.

Ew.d Floating point. The field is to occupy w positions. The list item is real, one part of a complex value or double precision. The value of the list item is to appear as a decimal number, right-justified in the field in the form:

- . xxxxxxseee

in which xxxxxx are the d most significant digits of the mantissa, s is a plus or minus sign, and eee is the corresponding 3-digit exponent. If the exponent is positive, a plus will precede eee . If it is negative, a minus will be written. If the value of the list item is positive, the minus sign in front of the decimal point is omitted or replaced by a plus sign if the SP format has occurred. The minimum field width necessary to contain a number of this type, including the sign of the fraction, is $w = 6 + d + sf$; with $sf = \max(p, 0)$ where p is the scale factor. If the field width specified is larger than necessary to contain the number, it is blank filled on the left. If the field width is too small to express the value to be written, the field is filled with asterisks.

On input, **Ew.d** is equivalent to **Fw.d**.

Ew.dEe Specific floating point. This is a special **Ew.d** format used for output only. The
Ew.dDe output format is:

- . xxxxxx Esee or - . xxxxxx Dsee

where e specifies the number of exponent digits. If the exponent will exceed the number of exponent digits specified, the field is filled with asterisks.

Dw.d Double precision floating point. This is the same as **Ew.d** except that the list item is regarded as double precision. The exponent is generally a 3-digit number. Thus, $w \geq 6 + d + sf$.

pP Scale factor (see 5.3.6).

BN Blank. This code controls the interpretation of embedded or trailing blanks in
BZ numeric input fields. The BN code indicates that embedded blanks will be ignored in numeric input fields. The BZ code indicates embedded blanks will be treated as zeros in numeric input fields. Embedded blanks in numeric input fields are normally treated as zeros during format control. If a BN is encountered in the format, embedded blanks will be ignored in numeric input fields until the end of that format or until a BZ format is encountered.

S Sign. This code controls the writing of optional plus signs in numeric output fields.
SP If SP or **+S** is used, all optional plus signs will be written. The field width must
SS be large enough to handle the sign or the field will be filled with asterisks. An SS, S, or **-S** turns the option off, and then only the minus signs will be written.

Lw Logical. This code is used only with input and output of logical variables. If **Lw** is specified for output and the value of the logical list item is **.TRUE.**, the rightmost position of the field with length w contains the letter T with $w-1$ blanks on the left. If the value is **.FALSE.**, the letter F is written, instead. On input, the field width is scanned from left to right for optional blanks, optionally followed by a decimal point, followed by a T or F, and the value of the corresponding logical list item is set to **.TRUE.** or **.FALSE.**, respectively. All other characters following the T or F in the external input field are ignored. In the absence of T or F in the input field, no value will be stored and a warning message will be issued.

- O w** Octal. The field is to occupy w positions, the value of the list item is to be interpreted as a 12-digit (or 24-digit) octal number, and the quantity is written as an octal number that is right-justified in the field. If the field width, w , is less than or equal to 12, then the w least significant digits appear. If w is larger than 12, it is filled out to the left with blanks on output and binary zeros on input. If the list item is double precision, 24 digits may be read or written. If w is less than or equal to 12, a double precision list item is treated as a single precision item; only the most significant word of the list item is used. If the output list item is character, the leftmost w characters of the list item are placed in the field with blank fill on the left if w is greater than the character item length. If the input list item is character, the leftmost portion of w characters will be stored to the list item with blank fill on the right if w is less than the length of the list item.
- A[w]** Alphanumeric. The field is to occupy w positions. Let s be the length of the character list item. This is the length of the variable for type character, four for integer, real, or logical; eight for double precision. On output, if w is less than or equal to s , the leftmost w characters of the list item are placed in the field. If w is greater than s , then the s characters of the list item are placed right-justified in the field and the field is blank filled on the left. On input, if w is less than s , the w characters of the field are placed in the leftmost w characters of the list item and the rightmost $s-w$ characters are blank filled. If w is greater than or equal to s , the rightmost s characters are placed in the list item. If w equals zero, no characters will be transferred. If w is missing, the number of characters transferred will be the character length of a list item of type character only. The w is optional for list items of the type character only.
- R w** Right-justified alphanumeric. This code is similar to A w . However, if w is less than s on input, the next w characters are placed right-justified in the list item with zero fill (000, ASCII NUL). If w is less than s on output, the rightmost w characters in the list are transmitted.
- $wHh_1...h_w$** Hollerith. The field is to occupy w positions. The field consists of literal characters and is to be filled with the w characters (including blanks) which follow H. The field designation is independent of the list items. The length w is limited only by the external medium and by the maximum of 511 significant characters for w . On input, the Hollerith specification is read into the format specification itself and is available on the next write using the same nonvariable format. Variable formats cannot save the Hollerith data in the format itself.
- ' $h_1h_2...h_w$ '** Literal. This is the same as the Hollerith specification. The only difference involves apostrophes within the literal. If an apostrophe is to be included in the literal, two apostrophes must be used to indicate its position.
- wX** Skip. A field whose length is w is to be skipped. The field designation is independent of the list items. Skipping forward over a character position which has not yet been set in the record causes the character position to be set to blank. If w is negative the skipping will be in a backward direction. However, a negative w cannot go back further than the beginning of the record; the skipping stops at the beginning of the record. No blanking is done if w is negative.

G *w.d* General. If the output item (*M*) is real, an E or F editing code is used depending on the absolute value of *M*. If $10^{i-1} \leq |M| < 10^i$ and $d \geq i \geq 0$, the output field is formatted by $F(w-4).(d-i),4X$. If either of the above conditions is not satisfied, the output field is formatted by $Ew.d$. If the scale factor *p* (see 5.3.6) is to be applied, the editing code $pPEw.d$ is used. Note that the G editing code does not change the value of the item. Values are strictly numeric or logical. No conversion is done for character type for output. A scale factor of 0 is assumed when the F editing code is used. On input, the G editing code is the same as an F editing code without a scale factor. Thus, $Fw.d$ is used on input when $pPGw.d$ is specified. The G editing code also provides for integer, double precision, and logical conversion as though it were Iw , $pPDw.d$, and Lw , respectively.

G *w.dEe* General. This is a special $Gw.d$ format used for output only. The output format is:

$\pm .xxxxxx Esee$

where *e* specifies the number of exponent digits. This format will be used if the absolute value of the output item requires an E editing code. If the F format is used, the output field is formatted by $F(w-(e+2).(d-i),(e+2)X$.

T *w* Character position. This code causes the input or output operation to begin at the *w*th position of the record. Therefore:

FORMAT(10X,F10.3)

is equivalent to:

FORMAT(T11,F10.3)

On output, if *w* is greater than any character position written up to that time, the T editing code will blank the character positions between *w* and the highest character position written. The order of the associated list does not need to be the same as that of the record input. For example:

FORMAT(T50,F10.3,T5,F10.2)

is valid for writing the first list item in positions 50-59 and the second in positions 5-14. The first character position of a record is number one.

TL *w* Character position Left. This code causes the character position in the record to be moved *w* positions to the left or backward from the current position. If the current position is less than or equal to *w*, the new character position will be the first character position in the record.

TR *w* Character position Right. This code causes the character position in the record to be moved *w* positions to the right or forward from the current position. The TR *w* cannot write beyond the end of the record.

If numeric output values fail to fit in the specified field length, the field will be filled with asterisks and no error will be given.

Examples of the uses of these editing codes follow:

```
C           Integer codes:
K=76
M=3333
WRITE (6,10) K
10  FORMAT(1X,I10)
WRITE(6,20) K
20  FORMAT(1X,I10.7)
WRITE(6,30) K
30  FORMAT(1X,J10)
WRITE (6,40) M
40  FORMAT (1X,I3)
C           With format 10, integer K will be written as ΔΔΔΔΔΔΔΔ76
C           With format 20, K will be written as ΔΔΔ0000076
C           With format 30, K will be written as 0000000076
C           With format 40, M will be written as *** since M does not
C           fit in the 3-digit field.
```

```
C           Real value codes:
R=76.8
WRITE(6,40) R
40  FORMAT(1X,F7.2)
WRITE(6,44) R
44  FORMAT(1X,E10.2)
WRITE(6,46) R
46  FORMAT(1X,D11.2)
WRITE(6,48) R
48  FORMAT(1X,SP,G10.4)
C           Format 40 writes R as ΔΔ76.80.
C           Format 44 writes R as ΔΔΔ.77+002.
C           Format 46 writes R as ΔΔΔΔ.77+002.
C           Format 48 writes R as +76.80ΔΔΔΔ
```

```
C           Alphanumeric editing codes:
CHARACTER*4 X,Y
CHARACTER*8 A,B
50  FORMAT(A3,A5,R2,R6)
READ(5,50)A,X,Y,B
55  FORMAT(1X,A6,A3,R5,R3)
WRITE(6,55)X,X,Y,Y
END
```

```
C           If the string 'ABCDEFGH IJKLMNOP' was provided for
C           input, the READ statement would produce
C           the following values:
C           A='ABCΔΔΔΔΔ'
C           B='■ ■KLMNOP'
C           X='EFGH'
C           Y='■ ■IJ'
C           The WRITE statement will produce the record:
C           ΔΔEFGHEFGΔ■ ■ IJ■ IJ
C           Note that in these examples, Δ represents the
C           nonprinting graphic for code 040 (the ASCII space
C           code) and ■ represents the nonprinting graphic
C           for code 000 (the ASCII NUL).
```

5.3.2. Editing Code Repetition

Whenever two or more successive editing codes (except *wH*, *wX*, *Tw*, *TLw*, *TRw*, */*, *S*, *SP*, *SS*, *BN*, *BZ*, *pP*, *:*, and literals) are identical in every respect, a shorthand notation may be used. This is accomplished by writing the editing code only once and prefixing it with an unsigned integer constant less than 512 which indicates the number of repetitions of the field. For example, `FORMAT(I5,I5,I5)` could be written as `FORMAT(3I5)`.

5.3.3. Repetition of Groups of Editing Codes

If two or more successive groups of editing codes occur and are such that each element of one group is identical to the corresponding element of the other groups, then a further shorthand notation may be used. This is accomplished by specifying the group only once, enclosing it in parentheses, and prefixing the resulting parenthesized group with an unsigned integer constant less than 512 which indicates the desired number of repetitions of the group. If none is specified, a repeat count of one is assumed.

For example, `FORMAT(I5,F9.4,E8.2,I5,F9.4,E8.2)` could be written as `FORMAT(2(I5,F9.4,E8.2))`. Only 4-deep nesting of parentheses is permitted (see 5.3.8). For example:

```
FORMAT('0',5(F9.3,5(E12.6,2(I8,2(I10,I5) ) ) ) )
```

is legal. However, if it had another nested group, that group would be executed as if it had a group count of one. The outside parentheses of the `FORMAT` statement are called the zero level parentheses.

If the list is longer than the number of items in the format, the format scan returns to the first level of parentheses (unless the format contains an indefinite repetition group). In the example, the scan would return to `F9.3` and continue until the input/output list was exhausted. Reversion of format control has no effect on scale factor, sign control, or blank control.

5.3.4. Carriage Control

In an output record to be printed, the first character of the record controls line spacing and is not printed. The line will start in the first character position with what would normally be the second character of the record. An appropriate first character is normally provided by the user with a Hollerith literal, literal, or an *wX* (which provides a blank) format. The effect of these carriage control characters is as follows:

<u>Character</u>	<u>Effect</u>
blank	Single space before printing
0 (zero)	Double space before printing
1	Skip to top of next page
+	Suppress spacing

If a Hollerith constant, literal, or *wX* format is not provided, the first character of whatever field comes first is used for carriage control. If it is an illegal character, the default blank character is used.

If an empty format is used without an input/output list, one record will be skipped on input or one blank record would be written on output.

5.3.5. Complex Variables

One complex variable requires two D, E, F, or G editing codes. They need not be the same. Both the real and imaginary parts of the variable must be written in the same record.

When read or written under control of a FORMAT statement, a complex value appears as a pair of real values. They are not enclosed in parentheses or separated by a comma unless the format specification provides them explicitly.

5.3.6. Scale Factor

Input and output using the F, E, G, and D editing codes may be modified by a power of 10 by using a scale factor of the form pP , where p is a signed integer.

The separating comma may be omitted between a scale factor and an F, E, D, or G editing code which immediately follows.

When format control is initiated, a scale factor of zero (that is, $0P$) is established. Once established, a scale factor applies to all subsequent F, E, G, and D editing codes within its FORMAT statement or until another scale factor is encountered within the statement. The scale factor does not affect any of the other editing codes.

For input, a scale factor, P , has the following effect:

- The scale factor has no effect on input with F, E, G, and D conversions that contain an exponent in the external field.
- For F, E, G, and D conversions with no exponent in the external field, the internal value is the external value divided by 10^p .

For output, a scale factor, P , has the following effect:

- For F conversions, the external value is the internal value multiplied by 10^p .
- For values using E or D conversions, the mantissa is multiplied by 10^p and the exponent is decreased by p . In other words, the field is changed in form but not in value on printed output.
- For G conversions, the effect of the scale factor depends on the magnitude of the value. If its size requires an E conversion, the scale factor effect is that of an E conversion. If an F conversion is required, the scale factor has no effect.

For instance, assume that variable A has the value -12764.316 . If A were printed according to the specification $E13.6$, then the field printed would be $-.127643+005$. If A were printed according to the specification $2PE14.6$, then the field printed would be $-12.764316+003$. If A were printed according to the specification $-3PF7.2$, then the field printed would be -12.76 .

5.3.7. Control of Record Handling and List Fulfillment

The colon and slash delimiters allow the user to halt unnecessary output and to begin processing of a new record, respectively.

5.3.7.1. Multiple Line Formats

A slash following an editing code terminates the record being input/output. Format interpretation of a new record continues with the editing code following the slash for the next list item. If n slashes are written in sequence before the first editing code in the format, n blank records are written or n records would be skipped before reading a record. If n slashes are written after the last editing code in the format, n blank records are written or n records are skipped on input. If editing codes are used before and after n sequential slashes, $n-1$ blank records are written or $n-1$ records are skipped on input.

Separating commas may be omitted immediately before or after a slash.

The following examples would produce three blank lines if the output goes to the printer:

```
(///12)
(12///)
(1X///12)
(12///1X)
(12///F8.2)
```

The third and fourth examples place one blank in a record or line which is then printed to get three blank lines.

If the file is a direct access file, the record number will be increased by the number of records skipped on input or by the number of blank records written on output.

Slashes may also be used in list-directed input/output (see 5.5).

If a single slash is used as the format, that is, (/), without an input/output list, two records would be skipped on input or two blank records would be written on output. An empty format, that is, (), without an input/output list will cause one record to be skipped on input or one blank record to be written on output.

5.3.7.2. End of Input/Output List Test

A colon in the format item list indicates that when a record is being read/written and the colon is encountered, the operation is ended if the last item in the list has already been processed. A single colon may appear at the beginning or end of the FORMAT statement or between pairs of format items.

Separating commas may be omitted immediately before or after a colon.

An example of the use of the colon for output is:

```
1  FORMAT ('ΔA Δ=', F10.3, ', ΔBΔ=', F10.3, ' ΔCΔ=', F10.3)
   IF (I.EQ.1) PRINT 1, A
   IF (I.EQ.2) PRINT 1, A, B
   IF (I.GT.2) PRINT 1, A, B, C
```

If the values supplied are I=2, A=17., and B=3.1416, the output will be:

```
ΔA Δ=ΔΔΔΔ 17.000ΔBΔ=ΔΔΔΔΔ 3.142
```


If commas had been used in place of the colons, the output would have printed C= before discovering the list is empty, resulting in:

```
ΔA Δ=ΔΔΔΔ 17.000ΔBΔ=ΔΔΔΔΔ 3.142ΔCΔ=
```

An example of the use of the colon on input is:

```
2  FORMAT (F10.3 / )  
1  FORMAT (F10.3, : / )  
   READ (5,2) E  
   READ (5,1) E1
```

If the input is:

```
ΔΔΔΔΔΔΔ 1.2  
ΔΔΔΔΔ 104.2  
ΔΔΔΔΔΔΔΔΔ 2  
ΔΔΔΔΔΔΔ 111
```

The first READ statement sets E to 1.2. The second READ statement sets E1 to .002. If the first READ statement had used a colon before the slash, E1 would be set to 104.2.

5.3.8. Relationships of a Format to an I/O List

During the execution of an input/output statement, the format specification is scanned from left to right. Editing codes of the form *w*H, *w*X, *p*P, BN, BZ, S, SP, SS, TL*w*, TR*w*, and T*w* as well as slashes and literals are interpreted, and the appropriate action is taken without reference to the input/output list. When an editing code of any other form occurs, either there is at least one list element remaining to be transmitted or there is not. If one remains, the next list element is converted according to the specification and transmitted. The format scan then continues. If no values remain, the transmission is terminated. (See 5.3.7.2 for the effect of a colon in the input/output list.)

There may be up to five levels of parentheses in a FORMAT statement. The outermost pair is called the 0-level parentheses and the innermost pair is called the 4-level parentheses. If, during the course of the format scan, the end of the FORMAT statement is reached while there is at least one more list element to be transmitted, a new record (for example a line, a card, or a tape record) is started, and the scan is resumed with the group repeat specifications preceding the rightmost 1-level left-hand parenthesis. In the example shown in 5.3.3, the second (and every following) record is started with a F9.3 editing code. If there are no nested parentheses, scan control reverts to the beginning of the format specifications. It is illegal to specify more than 80 characters for one record of symbiont input or more than 132 characters for a symbiont output record unless an OPEN statement specified a different record size for the symbiont. If the scale factor has been modified by an *p*P specification, it is not reset to zero when control reverts to the appropriate left parenthesis. Each *p*P specification remains in effect until the end of the FORMAT input/output list or until another *p*P specification is encountered (see 5.3.6).

If the format specification is empty and an I/O list is present, list-directed I/O will be used. If the I/O list is not present and the format specification is empty, one record will be skipped on input or one blank record will be written on output.

5.3.9. Variable Formats

Any of the input/output statements which specify formats may contain an array or scalar character variable name or a character expression in place of the reference to a format.

Let the symbolic name *V* be an array. The contents of the array may be initialized and modified like any array. Array *V* must contain legal format specifications in ASCII form at the time it is used as a format. The format specifications in *V* must be of the form:

(format-specification)

that is, the form of this specification is exactly as described in 5.3 except that the word **FORMAT** must be omitted. Blanks may appear before the first left parenthesis (0-level), indicating the beginning of the format. Characters following the right parenthesis, which indicates the end of the format (0-level right parenthesis), are ignored. For example:

```
CHARACTER*4 V
DIMENSION V(5), K(8)
DATA (V(I), I=1, 5) / 4H(1H1, 2H, 8, 1H1, 2H10, 1H) /
READ(5, 60) K
WRITE(6, V) K
60  FORMAT(8I10)
END
```

This will generate (1H1,8ΔΔ|ΔΔΔ 10ΔΔ)ΔΔΔ. The blanks (Δ) will be ignored when the array is used as a format. The use of *V* as a format is equivalent to using:

```
FORMAT(1H1,8I10)
```

Hollerith specifications inside variable formats must be of the form *nH*. The variable format is converted by the library to a coded format and stored in a library storage area. The variable format is not changed by the library. The variable format may be changed by the program. **If the H editing code is used in a variable format on a READ statement, the Hollerith editing code in the variable format will not contain the Hollerith data read in; that is, the variable format will not be changed.**

A character expression in the control list as the format could be:

```
WRITE(6, '(1X,8I10)') K
```

5.3.10. Representation of Input/Output Data

The format of data in an input field can be the same as that generated by the same format editing code for output. Several relaxations in format rules are, however, permitted:

- Unless positioning is achieved by the X, T, or TR editing codes, column 1 is the first column read. There is no carriage control.
- Plus signs may be omitted, or may be indicated by +. Minus signs are indicated by -.
- Embedded or trailing blanks in a numeric input field are equivalent to zeros unless the BN format is used or BLANK=NULL on the OPEN statement is used (see 5.10.1).
- Only the high order eight digits are retained for E, F, and G conversions, and the high order 18 digits are retained for D conversion.

- For E and D conversions, the exponent can be indicated by an E or a D (interchangeable), or, if E or D is omitted, by + or - (not a blank). Thus, E2, E+2, E+02, +2, and D2 are all permissible, equivalent exponents. Lowercase e and d are recognized as E and D.
- For E and D, if the exponent is omitted altogether, it is taken to be either zero or the value of the P specification (see 5.3.6).
- Numbers for E, D, F, and G conversions need not have the decimal point; the *d* field of the editing code implies it. When a decimal point appears in the field, it overrides the *d* specification.
 - The exponent is right-justified in the subfield defined by the E or D, or +, or -, at the right-hand limit of the field. The subfield may be empty. The mantissa is right-justified in the remaining field, to the left of the E, D, +, or -.
 - If the decimal point is omitted, it is assumed to be located between the *d* and *d*+1 positions (from the editing code) to the left of the exponent. If no exponent is present, the decimal point is assumed to be located between the *d* and *d*+1 rightmost positions in the field.

5.4. Namelist

The nonexecutable NAMELIST statement and the associated forms of the input/output statements provide a simplified means of transmitting an annotated list of data to and from files.

The items to be transferred are specified in the NAMELIST statement. In the associated input/output statement, the list of items to be transferred is void. See 5.6.1.3 and 5.6.2.3 for details on input/output statements which use the namelist specification. Therefore, by referring to a single name *n*, it is possible to form a simple input/output statement which has the same effect as a similar statement with a long list and a reference to a complicated FORMAT statement.

On input, the list of items in the namelist specifies those items which may have their values defined in the records to be read. Not all items of the namelist need be used in the input records nor must the input fields be in the same order as the list items. If a list item is an array name, data may be assigned to the entire array, a group of contiguous elements, or any individual elements of the array as specified by the input records. Within input records, the data to be read is identified within the input record itself. Thus, to read the number 1.5 into variable A, the external input field contains:

A=1.5

where the field is delineated by commas. The acceptable forms of the data and the format in which the data is to be stored are dictated by the type of the list item (see 5.4.2).

On output, each list item of the designated namelist is formatted in a standard fashion for output in the order specified by the list (see 5.4.3). Namelist output may be used as a convenient source language diagnostic tool, especially when it is coupled with a parameterized DELETE statement.

5.4.1. NAMELIST Statement

Purpose:

The NAMELIST statement provides a simplified means of identifying a list of data for transferral to or from files.

Form:

```
NAMELIST /n/x[ ,x] ... [/n/x[ ,x] ... ] ...
```

where:

each n is a namelist name which must satisfy the rules for a symbolic name. Each must be unique within its program unit.

each x is a simple variable, a subscripted variable, or an array name.

Description:

Within a NAMELIST statement, a namelist name is enclosed in slashes. The list of variables associated with a namelist name ends when a new namelist name enclosed in slashes is encountered or with the end of the NAMELIST statement.

A namelist name may be defined only once in a program unit by its appearance in a NAMELIST statement. Within the unit in which it is defined, a namelist name may appear only in input/output statements and in the defining NAMELIST statement.

A variable name, array element name, or array may be associated with one or more namelist names. Array names must have been previously declared. The subscripts of an array element must consist of constants or parameter constants. Dummy arguments are not permitted in a namelist list. A \$ may not be used in the variable name, array element name, or array name.

For a description of local-global rules for names used in NAMELIST statements in internal subprograms, see 7.11.

See 5.4.2 for details on input associated with a NAMELIST statement and 5.4.3 for information on the form of output resulting from the use of a namelist WRITE statement.

Example:

```
DIMENSION A(10), I(5,5), L(10)
NAMELIST/NAM1/A,B,I,J,L(3)/NAM2/A,C,J,K,I(2,3)
C           Arrays A and I, variables B and J, and subscripted
C           variable L(3) are associated with namelist name NAM1,
C           and array A and variables C, J, K, and I(2,3) are
C           associated with namelist name NAM2.
```

5.4.2. Namelist Input

The READ (u,n) statement (see 5.6.1.3) causes the records that contain the input data for the variables and arrays that belong to the namelist name n to be read from input unit u .

For READ (u,n), the first character of the first record of a group of data records to be read is always ignored, and the second character must be the currency symbol (\$) or ampersand (&), immediately followed by the namelist name and a blank. If the name is not n , the input medium is searched for \$ n . The remainder of the first record and following records may contain any combination of the legal data items which are separated by commas (a comma after the last data group is ignored). The last input record is terminated by the characters \$END or &END.

The forms the input data items may take are:

- *variable-name = constant* (*Variable-name* is a simple variable name.)
- *subscripted-variable = constant* (The array element appears in the NAMELIST statement. The subscripts in the input record must be constants.)
- *subscripted-variable = set-of-constants* (The set is represented by constants separated by commas, where *k*constant* represents *k* constants. The name of the array of which the subscripted variable is an element must be in the namelist list. The number of constants must not exceed the space available in the array, which is from the element specified to the end of the array, inclusively.)
- *array-name = set-of-constants* (The set is represented by constants separated by commas, where *k*constant* represents *k* constants. The number of constants must be less than or equal to the number of elements in the array.)

Constants used in the data items may take any of the following forms:

- Integer constants (see 2.2.1.1).
- Real Constants (see 2.2.1.2).
- Double precision constants (see 2.2.1.2.2). The D is optional in the exponent.
- Complex constants (see 2.2.1.3).
- Logical constants (see 2.2.1.4). These may be written as .TRUE., .FALSE., T., or F. The value stored is 0 for false and 1 for true.
- Character constants (see 2.2.1.5). Character input must not span records. The full *nH* must appear on one line as well as the full literal.
- Octal constants (see 2.2.1.6). If the type of the namelist variable is double precision, 24 digits may be used.

Logical and complex constants may be associated only with logical and complex variables, respectively. Hollerith constants may only be associated with integer or real variables. Literals may be associated with character variables only. The other types of constants may be associated with integer, real, or double precision variables, and they are converted in accordance with the type of the variable.

The data items that appear in the input records need not appear in the same order as the corresponding variable or array names in the NAMELIST statement list. All variable or array names of the namelist need not have a corresponding data item in the input records. If none appears, the contents of the variable or array are unchanged. Names that are equivalenced to these names may not be used in the input records unless they also are part of the namelist.

Blanks must not be embedded in a constant or a repeat constant field, but may be used freely elsewhere in a data record. The last item in each record that contains data items must be a constant, optionally followed by a comma. The comma is optional in the record that contains or precedes the \$END sentinel.

For example, the NAMELIST statement:

```
DIMENSION A(10), K(5,5), L(10)
NAMELIST/NAM1/L(3),A, K, J
```

could have the following input records:

<u>Record</u>	<u>Position and Contents</u>
First Data Record	Δ\$NAM1Δ K(2,3)=5,7,9,
Second Data Record	0,10,L(3)=4.3,A=4.3,
Third Data Record	8*4.3,J=4.2,
...	...
Last Data Record	\$END

If the preceding data is used with a READ, the following action takes place: the first data image is read and examined to verify that its name is the same as the namelist name in the READ statement. (Note that the name must begin in column two of the first record.) The integer constant 5 is placed in K(2,3), 7 in K(3,3), and 9 in K(4,3). When the second data image is read, integer 0 is placed in K(5,3), and 10 in K(1,4); the real constant 4.3 is truncated to integer 4 and placed in L(3); integers 4 and 3 are converted to real and placed in A(1) and A(2), respectively. When the third data image is read, the real constant 4.3 is stored in A(3), A(4), . . . , A(10); and the real constant 4.2 is truncated to integer 4 and placed in J.

5.4.3. Namelist Output

The WRITE (*u,n*) statement (see 5.6.2.3) causes all names of variables and arrays (as well as their values) that belong to the namelist name *n* to be written to the output unit *u*.

In the WRITE (*u,n*) statement, all variables and arrays (as well as their values) belonging to the namelist name are written out according to their types. The output data is written such that:

- The name of a variable and its value are written on one line or record.
- The name of an array is written, with the values of the elements of the array written in a convenient number of columns, in the order of the array in main storage, that is, in column-major order.
- Whole arrays will always start at the beginning of the output buffer (that is, column 1).
- The data fields are large enough to contain all the significant digits.
- The output can be read by an input statement referring to the namelist name.

For example, if the values of the variables A through H and arrays L and M have the values 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0; L(1) = 5 and M = 1,2,3,4, the statements:

```
DIMENSION L(4), M(2,2)
NAMELIST /OUT1/ A,B,C,D,E,F,G,H,L(1),M
WRITE(8,OUT1;
```

would cause the following five records to be written (where a record represents one line printed):

```
$OUT1
A = .10000000+001, B = .20000000+001, C = .30000000+001, D = .40000000+001, E = .50000000+001,
F = .60000000+001, G = .70000000+001, H = .80000000+001, L(1) = 5,
M = 1, 2, 3, 4
$END
```

5.5. List-Directed Input/Output

List-directed input/output statements are used to read and write free-field records. A list-directed write will output records consisting of sequences of values composed of characters which can be represented internally. The order of the values is the order in which they are listed.

A list-directed read will input as many records as necessary to provide a value for each item in the input list, unless a slash appears in the input. In this case, the slash serves as an end-of-record-transmission signal. Any list items remaining after the slash has been encountered undergo no change in value (see 5.5.1).

The output from a list-directed write (except for a write to a print or punch unit) can be used as input to a list-directed read unless the input variable is character.

5.5.1. List-Directed Input

The list-directed input consists of a sequence of values separated by commas, blanks, slashes, or end of the line. Numeric, logical, and character data may be read. Hollerith and octal data may not be used.

If a value is to be repeated for several list items, it can be written as:

$$r * c$$

where r is an unsigned, nonzero, integer constant and c is a value that may be blank. Blanks may not be embedded in either r or c , except within character constants and complex constants. Examples of this form are:

$$5 * 4.0$$
$$5 * \Delta$$

Each constant must be of the same type as its corresponding list item. An error occurs if the types do not match. Blanks are never used as zeros and embedded blanks are not permitted except within character constants and complex constants. For example, $5, \Delta 0, 2.0$ represents three values (5.0, 0.0, and 2.0) and $5.0, \Delta 2.0$ represents two values. The end of a line may be considered a blank except when it appears in a character constant.

A null value is specified by:

- no characters between two commas (, ,)
- no characters before the first comma in the list-directed input record
- only blanks between two commas (, ΔΔ ,)
- only blanks before the first comma in the list-directed input record
- a blank following the asterisk in the r*c form (r * Δ)
- a comma following the asterisk in the r*c form (r * ,)

A null value has no effect on the corresponding list item. If the list item has no value, it will still have no value. A null value may not be used as the real or imaginary part of a complex constant, but it may represent an entire complex constant. The end of a line before or after a comma does not generate a null item.

A slash causes the end of the list-directed read after the assignment of the previous value. If there are more items in the list, the effect is as though null values are supplied for them.

The parts of a complex constant must be separated by a comma and enclosed in parentheses. The comma may be preceded or followed by spaces or the end of a line. For example:

```
(25.2,  
 2.0)
```

and:

```
(25.2, ΔΔΔ 2.0)
```

are legal forms of a complex constant.

Hollerith constants are not allowed in free-field input records, but character constants in apostrophes are legal if the type of the list item is character. Each apostrophe within a literal must be represented by two apostrophes with no intervening spaces or end of line. Character constants may be continued from one line to the next with no blank inserted for the end of the line. Character constants may contain blanks, commas, or slashes.

5.6. Sequential Access Input/Output Statements

There are five sequential access input/output statements:

- READ (reads records from a sequential file)
- WRITE (writes records to a sequential file)
- BACKSPACE (positions file for input/output by backspacing one record)
- ENDFILE (defines the end of a sequential file)
- REWIND (positions a unit at its beginning)

The OPEN statement and the DEFINE FILE statement are used in conjunction with these statements. They allow the programmer to describe the format of a file.

Using these statements, a file may be processed in sequential order. Each record is manipulated or passed over, as desired, one after another. Unlike direct access input/output statements which refer to random access files, sequential access input/output statements cannot directly address a specific record.

5.6.1. READ Statements

READ statements obtain values for data elements from files. The form of each READ statement depends upon the type of record (formatted, unformatted, namelist, or list-directed) being read.

5.6.1.1. Formatted READ

Purpose:

A formatted READ statement reads values into items specified in an input list according to a specified format.

Form:

```
READ ([UNIT=] u, [FMT=] f [, ERR=s] [, END=sn] [, IOSTAT=ios]) [iolist]
```

or:

```
READ f [, iolist]
```

where:

[UNIT =] *u* is a file reference number specification which must be present (see 5.2.1); *u* is a file reference number. The UNIT= clause is an optional part of the specification.

[FMT =] *f* is a format specification (see 5.2.4); *f* must be present but the FMT= clause is an optional part of the specification.

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

`END = sn` is an optional end clause specification (see 5.2.7); `sn` is a statement label.

`IOSTAT = ios` is an optional I/O status clause specification (see 5.2.8); `ios` is an integer variable or integer array element.

`iolist` is an input list (see 5.2.3).

Description:

The formatted READ statement causes one or more formatted records to be read from the file specified by `u`. The information in the records is scanned and converted as specified by the format specification `f`. The resulting values are assigned to the variables specified in the `iolist`. If the optional `UNIT=` and `FMT=` clauses are used, the specifications may appear in any order. If the `UNIT=` clause is used, the `FMT=` clause must be used.

The second form which does not have a file reference number specification implies that the symbiont file `AREAD$` is to be used. This is equivalent to specifying unit number 5 (the `AREAD$` symbiont, see G.6) in the first form.

Examples:

```
10 READ 10, IVAR1, ARRAY2
   FORMAT(18,10E9.2)
```

```
C This formatted READ statement reads (from the
C program input unit) an integer value into variable
C IVAR1 and 10 real values into array ARRAY2.
```

```
20 READ (3, 20, ERR = 140) IVAR1, ARRAY2
   FORMAT(18, 10E9.2)
```

```
C This formatted READ statement obtains values from input
C unit number 3 and assigns them to members of the input
C list according to the format specified in statement 20.
C If an error occurs during processing of this statement,
C program control is transferred to statement 140.
```

```
30 READ (FMT = 30, UNIT = 3, IOSTAT = IVAL) VAR1, ARRAY3
   FORMAT(F8.2,8E9.2)
```

```
C This formatted READ statement obtains values
C from input unit 3 and assigns them to members of the input
C list according to the format specified in statement 30.
C The order of the clauses is optional when the UNIT =
C and FMT= clauses are used. If an error or end-of-file
C condition occurs during the processing of this statement,
C a nonzero value (see 5.2.8) will be stored in IVAL and
C control will be returned to the user after the READ
C statement. If the statement is successful, a zero
C will be stored in IVAL.
```

5.6.1.2. Unformatted READ

Purpose:

The unformatted READ statement reads values into items specified in an input list, with no conversion or reformatting of the input values.

Form:

```
READ ([ UNIT=] u [, ERR=s] [, END=sn] [, IOSTAT=ios]) [iolist]
```

where:

[UNIT =] *u* is a file reference number specification which must be present (see 5.2.1); *u* is a file reference number. The UNIT= clause is an optional part of the specification.

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

END = *sn* is an optional end clause specification (see 5.2.7); *sn* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

iolist is an input list (see 5.2.3).

Description:

The unformatted READ statement causes the next record to be read from the file specified by *u*. The record values are assigned to the variables specified by the *iolist* with no conversion. The number of values required by the *iolist* must be less than or equal to the number of values in the unformatted record. If the optional UNIT= clause is used, the specifications in the control list may appear in any order.

Examples:

```
      READ (3) VAR2, VAR3, ARRAY3
C      This statement obtains values from input unit 3 and,
C      without changing their forms, assigns values to variables
C      VAR2 and VAR3 and to the elements of array ARRAY3.
```

```
      READ (3, ERR = 150, END = 280) VAR2, VAR3, ARRAY3
C      This statement is executed in the same way as the first
C      example. However, if an error is detected during
C      statement execution, control is transferred to the
C      statement labeled 150 rather than stopping execution.
C      If the input file is currently positioned at end-of-file,
C      program control will be passed to the statement labeled 280.
```

```
      READ (END = 280, ERR = 150, UNIT = 3) VAR2, VAR3, ARRAY3
C      This statement is equivalent to the previous statement.
C      The optional UNIT= clause was used so that the parameters
C      within the control list may appear in any order.
```

5.6.1.3. Namelist READ

Purpose:

The namelist READ statement provides a simplified means of obtaining an annotated list of data from an input device.

Form:

```
READ ( u, n [ , ERR = s ] [ , END = sn ] [ , IOSTAT = ios ] )
```

or:

```
READ n
```

where:

- u* is a file reference number which must be present (see 5.2.1).
- n* is a namelist name specification (see 5.2.5).
- ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.
- END = *sn* is an optional end clause specification (see 5.2.7); *sn* is a statement label.
- IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

The namelist READ statement allows reading of specially formatted records without specifying an *iolist* on the READ statement. See 5.4 for further details.

The second form does not have a file reference number specification. It implies that the symbiont file AREAD\$ is to be used. This is equivalent to specifying unit number 5 in the first form (see G.6).

Example:

```
      DIMENSION ARR4(10), ARR5(4,5)
      NAMELIST/SPECN1/VAR3, VAR4, ARR4, ARR5, VAR5, VAR6
      READ (5, SPECN1, ERR = 140)
C          Data for the variables specified in the namelist called
C          SPECN1 are read in.
```

5.6.1.4. List-Directed READ

Purpose:

The list-directed READ statement reads in specially formatted records without the user having to specify an associated FORMAT statement.

Form:

```
READ ([UNIT =] u, [FMT =] * [, ERR = s] [, END = sn] [, IOSTAT = ios])[iolist]
```

or:

```
READ * [, iolist]
```

where:

[UNIT =] *u* is a file reference number specification which must be present (see 5.2.1); *u* is a file reference number. The UNIT= clause is an optional part of the specification.

[FMT =] * is a format specification clause; * indicates a list-directed statement and must be present. The FMT= clause is an optional part of the specification.

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

END = *sn* is an optional end clause specification (see 5.2.7); *sn* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

iolist is an input list (see 5.2.3).

Description:

Execution of the list-directed READ statement causes data to be read from the file specified by *u*. These data values are assigned to the variables specified by *iolist*.

The second form implies use of the symbiont file AREAD\$.

A record containing list-directed input data consists of constants, null values, and value separators.

The ERR, END, and IOSTAT clauses may appear in any order.

If the UNIT= and FMT= clauses are present, the specifiers may appear in any order in the control list. If the UNIT= clause is used, the FMT= clause must be used.

See 5.5 for additional details.

Examples:

```
      READ *, VAR3, VAR4, ARR5, ARR6
C      Reads in formatted values for the specified variables
C      and arrays.
      READ (7, *, END = 260, ERR = 140) VAR1, VAR2
C      Reads formatted values from records of unit 7. If an
C      error is detected during execution of this READ
C      statement, control is passed to statement 140. If
C      fewer records exist in the input file than required
C      to fill the elements in the list, control transfers
C      to the statement labeled 260.
      READ (ERR = 140, FMT = *, END = 260, UNIT = 7) VAR1, VAR2
C      This statement is the same as the previous READ except that
C      the UNIT= and FMT= clauses allow the control list items
C      to appear in any order.
```

5.6.1.5. Reread

A file reference number may be assigned to reread the last formatted symbiont or SDF sequential record read. Thus, it is possible to read the last record any number of times. Input items read with editing codes according to their types could, for instance, be reread with an A editing code as alphanumeric information. Subsequent formatted READ statements change the record to be reread. The last record read will be unavailable if an OPEN or DEFINE FILE is executed which defines a record size larger than previously defined or defaulted record sizes.

The unit number zero (0) will be a default reread unit. Other units may be declared as reread units by generating a new file reference table (see G.6) or through the OPEN statement (see 5.10.1). However, only a unit which has been closed or never opened may be opened for rereading with an OPEN statement. The format of the reread statement is:

```
READ ([UNIT=] u, [FMT=] f [, ERR=s] [, IOSTAT=ios]) iolist
```

When rereading a record, the BLANK=NULL clause provided by the OPEN statement will be ignored. All blanks other than leading blanks will be treated as zeros unless the BN format occurs in the format if provided.

By using the reread capability, more than one attempt to read a record can be made. If an error results from a READ with one format, an ERR clause can direct control to another READ which can reread the record with a different format.

If, for example, 0 were the reread unit number and X and Y were formats, the following sequence could make three attempts to read a record:

```
      READ (5,X,ERR=10) list
      .
10     READ (0,Y,ERR=20) list
      .
20     READ (0,*,IOSTAT=IVAL) list
      IF (IVAL.NE.0) GO TO 50
      .
50     STOP 777
```

If the record could not be read with format X, format Y would be tried. If that attempt resulted in an error, the list-directed read would be used.

An END= clause will be ignored on a reread statement. The reread statement must not request more than one record.

5.6.2. Output Statements

WRITE statements transfer data from variables into files. For each type of READ statement, there is a corresponding output statement. Likewise, the various WRITE statement forms depend on the type of record (formatted, unformatted, **namelist**, or list-directed) being written.

5.6.2.1. Formatted WRITE

Purpose:

The formatted WRITE statement writes specified data into a specific file according to a given format.

Form:

```
WRITE ([UNIT=] u, [FMT=] f [,ERR=s] [END=sn] [,IOSTAT=ios] ) [iolist]
```

or:

```
PRINT f [, iolist]
```

or:

```
PUNCH f [, iolist]
```

where:

[UNIT =] *u* is a file reference number specification which must be present (see 5.2.1); *u* is a file reference number. The UNIT= clause is an optional part of this specification.

[FMT =] *f* is a format specification (see 5.2.4); *f* must be present but the FMT= clause is an optional part of the specification.

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

END = *sn* is an optional end clause specification (see 5.2.7); *sn* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

iolist is an output list (see 5.2.3).

Description:

The formatted WRITE statement causes one or more formatted records to be written to the file specified by *u*. The variables specified by the *iolist* are edited according to the format specification *f* and written to file *u*. If the optional UNIT= and FMT= clause are used, the order of the various specifications is optional. If the UNIT= clause is used, the FMT= clause must appear.

The PRINT form implies that the symbiont file APRINT\$ is to be used as the output file. This is equivalent to the WRITE form with a file number of 6 (the APRINT\$ symbiont, see G.6).

The PUNCH form implies that symbiont file APUNCH\$ is to be the output file. This is equivalent to the WRITE form with a file number of 1 (the APUNCH\$ symbiont, see G.6).

Examples:

```
WRITE(8, 40, ERR = 160) VAR1, VAR2, ARR1
```

C The contents of variables VAR1 and VAR2 and of array
C ARR1 are to be written into file unit 8 using the
C format specified in the statement labeled 40. If an
C error occurs during processing of this statement,
C control is to be transferred to the statement labeled 160.

```
WRITE (FMT = 40, UNIT = 8, ERR = 160) VAR1, VAR2, ARR1
```

C This is the same as the first WRITE except that with the
C presence of the UNIT= and FMT= clauses, the clauses in the
C control list may be in any order.

```
PRINT 40, VAR1, VAR2, ARR1
```

C Prints the same values on system printer.

```
PUNCH 40, VAR1, VAR2, ARR1
```

C Punches the same values on cards.

5.6.2.2. Unformatted WRITE

Purpose:

The unformatted WRITE statement writes a record consisting of the specified data to a given file.

Form:

```
WRITE ([UNIT=] u [,ERR=s] [,END=sn] [,IOSTAT=ios])[iolist]
```

where:

[UNIT =] *u* is a file reference number specification which must be present (see 5.2.1); *u* is a file reference number. The UNIT= clause is an optional part of the specification.

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

END = *sn* is an optional end clause specification (see 5.2.7); *sn* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

iolist is an output list (see 5.2.3).

Description:

The unformatted WRITE statement causes one record to be written to the file specified by *u*. The record contains the sequence of values specified by the *iolist*.

Examples:

```
WRITE (8) ARR3, VAR1, VAR2
C      Writes a record to file 8 without reformatting of
C      data in the output list.

WRITE (ERR=180,UNIT=8) ARR3, VAR1, VAR2
C      Same as the previous example except that control is passed
C      to the statement labeled 180 if an error is
C      encountered while executing this statement. With the
C      presence of the UNIT= clause, the clauses in the
C      control list may be in any order.
```

5.6.2.3. Namelist WRITE**Purpose:**

The namelist WRITE statement allows the programmer to write a particular list of variables and arrays without including the output list in the WRITE statement.

Form:

```
WRITE ( u, n [ , ERR=s ] [ , IOSTAT=ios ] )
```

or:

```
PRINT n
```

or:

```
PUNCH n
```

where:

u is a file reference number which must be present (see 5.2.1).

n is a namelist name specification (see 5.2.5).

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

One or more records are written to the file specified by *u*. The records consist of data values specified in the NAMELIST set named *n*.

The PRINT statement implies that the symbiont file APRINT\$ is to be used. This is equivalent to specifying unit 6 (the APRINT\$ symbiont, see G.6) in the first form.

Refer to 5.4 for additional details on NAMELIST.

Examples:

```
NAMELIST /SPECN2/ VAR1, ARR2, VAR3, ARR4, VAR5
WRITE (8, SPECN2)
C          Causes each name specified in namelist SPECN2 to be
C          written on a record with its associated values.

WRITE (8, SPECN2, ERR = 180)
C          Same as the previous example except that program
C          control will be transferred to the statement labeled 180
C          if an error occurs during statement execution.
```

5.6.2.4. List-Directed WRITE**Purpose:**

The list-directed WRITE statements write out specially formatted records without specifying an associated FORMAT statement.

Form:

```
WRITE ([UNIT = ] u, [FMT = ] * [, ERR = s] [, END = sn] [, IOSTAT = ios]) [iolist]
```

or:

```
PRINT *[, iolist]
```

or:

```
PUNCH *[, iolist]
```

where:

[UNIT =] *u* is a file reference number specification which must be present (see 5.2.1); *u* is a file reference number. The UNIT= clause is an optional part of the specification.

[FMT =] * is a format specification; * indicates the statement is list-directed and must be present. The FMT= clause is an optional part of the specification.

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

END = *sn* is an optional end clause specification (see 5.2.7); *sn* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

iolist is an output list (see 5.2.3).

Description:

Execution of a list-directed WRITE statement causes data to be written to the file specified by *u* (or to the assumed print or punch file).

The PRINT form implies that the symbiont file APRINT\$ is to be used. This is equivalent to specifying unit number 6 (the APRINT\$ symbiont, see G.6) in the WRITE form.

The PUNCH form implies that the symbiont file APUNCH\$ is to be the output file, which is equivalent to specifying unit number 1 (the APUNCH\$ symbiont, see G.6) in the WRITE form.

If the UNIT= and the FMT= clauses are present, the specifiers in the control list may be in any order.

If the UNIT= clause is used, the FMT= clause must be used.

See 5.5 for more information on list-directed input/output statements.

Examples:

```
WRITE (8, *, ERR = 180) VAR3, VAR4, ARR5, ARR6
C      Writes out formatted data values to output file 8.
C      If an error occurs during processing, program control
C      is transferred to the statement labeled 180.
```

```
WRITE (ERR = 180, UNIT = 8, FMT = *) VAR3, VAR4, ARR5, ARR6
C      This is the same as the previous WRITE except that
C      the UNIT= and FMT= clauses are present and the clauses
C      in the control list may be in any order.
```

```
PRINT *, VAR3, VAR4, ARR5, ARR6
C      Writes formatted data values from list to the output
C      print file APRINT$.
```

```
PUNCH *, VAR3, VAR4, ARR5, ARR6
C      Punches these same values on cards via the output
C      punch file APUNCH$.
```

5.6.3. BACKSPACE Statement

Purpose:

The BACKSPACE statement repositions the file pointer for files such as magnetic tape, disk and drum units by backspacing one record. System files (for example, AREAD\$, APRINT\$, and APUNCH\$) and files not created by PCIOS cannot be backspaced (see G.2.3). Backspacing over a list-directed record is also prohibited.

Form:

```
BACKSPACE ( [ UNIT = ] u [ ,ERR = s] [ ,IOSTAT = ios ] )
```

or:

```
BACKSPACE u
```

where:

UNIT = is an optional clause (see 5.2.1).

u is a file reference number (see 5.2.1).

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

The file specified by *u* is backspaced one record.

If the unit identified by *u* is already at its initial point, the BACKSPACE statement has no effect.

An endfile record is counted as one record.

If the UNIT= clause is present, the specifiers in the control list may be in any order.

Example:

```
45   FORMAT ( ... )  
.  
.  
   READ (IOSTAT=IVAL, FMT=45, UNIT=MTAPE) iolist  
.  
.  
   READ (MTAPE, 45) iolist  
.  
.  
   BACKSPACE MTAPE  
.  
.
```

```
READ (FMT=45,UNIT=MTAPE) iolist
```

```
C      The first READ refers to the first record on MTAPE,  
C      the second READ refers to the second record, and the  
C      third READ also refers to the second record on MTAPE  
C      because the file pointer has been backspaced one record.
```

5.6.4. ENDFILE Statement

Purpose:

The ENDFILE statement marks the end of a file.

Form:

```
ENDFILE ([ UNIT=] u [,ERR=s] [,IOSTAT= ios])
```

or:

```
ENDFILE u
```

where:

UNIT = is an optional clause (see 5.2.1).

u is a file reference number specifying the file to be demarcated (see 5.2.1).

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

An end-of-file record is written to the file specified by *u*.

System files (for example, AREAD\$ and APRINT\$) cannot have an end-of-file record written on them using this statement.

If the UNIT= clause is present, the specifiers in the control list may appear in any order.

Example:

```
ENDFILE MTAPE
```

```
C      This statement causes an endfile record to be  
C      written on MTAPE to mark the end of a file.
```

5.6.5. REWIND Statement

Purpose:

The REWIND statement repositions the file pointer for a file to its starting point.

Form:

```
REWIND ( [ UNIT = ] u [ ,ERR = s ] [ ,IOSTAT = ios ] )
```

or:

```
REWIND u
```

where:

UNIT = is an optional clause (see 5.2.1).

u is a file reference number (see 5.2.1).

ERR = *s* is an optional error clause specification (see 5.2.5); *s* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

Execution of this statement causes the file pointer for *u* to be positioned at the file's initial point. Tapes are rewound to the load point and mass storage files are positioned at logical address zero.

System files (for example, AREAD\$, APRINT\$ and APUNCH\$) cannot be rewound.

If the UNIT= clause is present, the specifiers in the control list may appear in any order.

Example:

```
55     FORMAT ( ... )
      .
      .
      DO 65 I = 1, 20
      READ (MTAPE,55)
      .
      .
65     CONTINUE
      REWIND MTAPE
```

C During execution of the DO loop, 20 records are read
C from MTAPE. MTAPE is therefore positioned at the
C start of its twenty-first record when the REWIND
C statement is encountered. MTAPE is then repositioned
C to its initial address.

5.6.6. Sequential Access DEFINE FILE Statement

Purpose:

The sequential access DEFINE FILE statement describes the characteristics of a file which is to be used during a sequential access input/output operation. The three basic types which can be defined are: symbiont, SDF, or ANSI. (See 5.7.1 for use of the DEFINE FILE statement with direct access input/output operations). This statement must precede all executable statements in the program unit which refer to the file being described. If a DEFINE FILE or OPEN statement is not used to describe a file reference number, the unit will default to SDF type unless designated as a symbiont file type in the file reference table (see G.6). The first DEFINE FILE statement encountered for a unit is the one used during execution. Subsequent descriptions for SDF and ANSI will have a validity check done on the requested type, and then are ignored. Subsequent descriptions for symbiont units have a similar validity check, and the record size may be changed. All of the capabilities provided by a DEFINE FILE statement are included in the capabilities provided by the OPEN statement.

A DEFINE FILE statement or an OPEN statement must be used when:

- ANSI interchange tape files are to be processed.
- sizes other than default block size, record size, or segment size are desired for SDF files.
- sizes other than the normal default record sizes are desired for symbiont files.
- a define-file-block is to contain the file description (see G.10).

Form:

For symbiont:

```
DEFINE FILE u(st, , rs)
```

where:

u is an unsigned integer constant or integer parameter variable specifying a file reference number (see 5.2.1).

st specifies the symbiont type. It is coded as one of the following:

APRINT for ASCII print symbiont
 APUNCH for ASCII punch symbiont
 AREAD for ASCII read symbiont
 APRNTA for ASCII print alternate symbiont
 APNCHA for ASCII punch alternate symbiont
 AREADA for ASCII read alternate symbiont

rs is an integer constant or integer parameter variable specifying the record size in characters. The ranges allowed for *rs* and the defaults if *rs* is not specified are as follows:

<u>Symbiont</u>	<u>Range</u>	<u>Default</u>
APRINT, APRNTA, AREADA	$0 < rs \leq 160$	132
AREAD	$0 < rs \leq 132$	80
APUNCH, APNCHA	$0 < rs \leq 80$	80

For SDF files:

```
DEFINE FILE u(SDF, , [rs] , [bs] , [ss] )
```

where:

- u* is an unsigned integer constant or integer parameter variable specifying a file reference number (see 5.2.1).
- rs* is an integer constant or integer parameter variable specifying the record size in words (see G.7 for the default record size). Record size (*rs*) must be specified when records (other than unformatted records) larger than the default size (33 words) are to be written to an SDF file. Record size (*rs*) specification need not reflect unformatted records as they are read and written in segments.
- bs* is an integer constant or integer parameter variable specifying the block size in words. The default block size is 224 words (see G.2.1.3).
- ss* is an integer constant or integer parameter variable in the range $0 < ss \leq 2047$ specifying the record segment size in words (see G.8 for the default segment size). Record segment size (*ss*) is the size that all records will be segmented at when they are written to the SDF file. It is also the size at which unformatted records will be processed as they are processed in segments.

If the record size is given and a proper segment size and block size are chosen, the amount of input/output overhead can be minimized (see G.2.1.2 and G.2.1.3).

For ANSI tape files:

```
DEFINE FILE u(ANSI , [rf] , [rs] , [bs] , [bo] )
```

where:

- u* is an unsigned integer constant or integer parameter variable specifying a file reference number (see 5.2.1).
- rf* specifies the record format as shown below. The default record format is U.

<u>rf Code</u>	<u>Records Represented</u>	<u>Record Size (rs) Considerations</u>
U	Undefined	Indicate maximum record size.
F	Fixed, unblocked	} Indicate exact record size.
FB	Fixed, blocked	
V	Variable, unblocked	} Indicate maximum record size, excluding four characters of record control.
VB	Variable, blocked	
VS	Variable, unblocked, segmented	} Indicate maximum record size excluding the 5-character record control segment.
VBS	Variable, blocked, segmented	

The VBS and VS forms specify the ANSI S format. The V and VB forms specify the ANSI D format. The F and FB specify the ANSI F format. The U specifies the ANSI U format.

rs is an integer constant or integer parameter variable specifying the record size in characters. The default record size is 132 characters.

bs is an integer constant or integer parameter variable specifying the block size in characters. The default block size is 132 characters. The block size (*bs*) for ANSI files is dependent on the record size (*rs*), the buffer offset (*bo*), and the record format (*rf*) as follows:

- For U or F record format:

$$bs = rs + bo$$

- For V record format:

$$bs = rs + 4 + bo$$

- For FB record format:

$$bs = n(rs) + bo$$

where *n* is the number of records per block.

- For a VB record format:

$$bs = rs + 4 + bo$$

- For VS or VBS record format:

The block size may be less than, equal to, or greater than the record size, but it must be large enough to contain any buffer offset and the five characters of control information appended to each record segment.

bo is an integer constant or integer parameter variable in the range $0 < bo \leq 99$, which is the number of characters specifying the buffer offset. The default for buffer offset is 0. It must be specified if an ANSI input file contains buffer offset information in the front of the data blocks. This parameter is not required for ASCII FORTRAN-produced ANSI tapes since no buffer offset information is appended to the data blocks.

Description:

As illustrated in the form section, this statement may have one of three general forms depending on the type of file format.

If an optional parameter is omitted, its preceding comma may be omitted only if it is a trailing comma (no other parameters follow it). If an optional parameter is omitted, default parameters are used.

The DEFINE FILE statement for symbiont files may be used to define a unit which is not already defined as a symbiont file in the file reference table. It may also be used to specify a record size other than the default record size of 132 characters for print symbiont files and 80 characters for a punch or read file. If the unit is already defined as a symbiont file in the file reference table, then the symbiont type on the DEFINE FILE statement must match its definition. In this case, the user is only specifying the maximum record size for the symbiont file. If the symbiont file is a print file and the record size is larger than 132 characters, an ER is done to the print symbiont to expand the image size for the file. Therefore, in order to specify a 160 character print record, the DEFINE FILE statement would be used as follows:

```
DEFINE FILE u (APRINT , , 160)
```

or:

```
DEFINE FILE u (APRNTA , , 160)
```

When the FORTRAN program is terminated, the primary print symbiont will be reset to the system default record size.

For the punch symbionts, a record size of over 80 characters is not allowed. If a record size of over 80 characters is specified, the record size for the punch file will default to 80 characters.

A table of default and maximum values for symbiont files is given in G.10.

Examples:

```
DEFINE FILE 3 (ANSI , FB , 50 , 205 , 5)
```

```
C      This statement defines file 3 to be in ANSI format  
C      with fixed, blocked records of 50 character length.  
C      The block size is 205 characters with a buffer offset  
C      of 5 characters.
```

```
DEFINE FILE 9 (SDF , , 45 , 336)
```

```
C      The SDF file 9 is to have records 45 words long  
C      and a block size of 336 words.
```

5.7. Direct Access Input/Output Statements

The direct access statements permit a programmer to read and write records randomly from any defined location within a direct file.

There are three direct access input/output statements:

- READ (reads records from a direct file)
- WRITE (writes records to a direct file)
- FIND (allows repositioning of a direct file during program execution; this allows overlapping of record searching with other activities).

The OPEN and DEFINE FILE statements are used in conjunction with these statements.

Using these statements, a programmer can go directly to any point in a direct file, process a record, and go directly to any other point without having to process all the records in between. This is done using the relative record number. The relative record number is unique and is the record's relative position within the file.

Direct files may contain formatted records, unformatted records, or both. When formatted records are used, a FORMAT statement must specify the form in which data is to be transmitted.

5.7.1. Direct Access DEFINE FILE Statement

Purpose:

The direct access DEFINE FILE statement describes the characteristics of a direct file that is used during a direct access input/output operation. This statement must logically precede any READ, WRITE or FIND statements pertaining to the file it describes. All of the capabilities provided by a DEFINE FILE statement are included in the capabilities provided by the OPEN statement.

Form:

```
DEFINE FILE u (m, s, b, v) [ , u(m, s, b, v) [ , ... u (m, s, b, v) ] ]
```

where:

- u* is an unsigned integer constant or integer parameter variable specifying a file reference number (see 5.2.1).
- m* represents an integer constant or integer parameter constant that specifies the maximum number of records the file will contain.
- s* represents an integer constant or integer parameter constant that specifies the maximum record size for a file. The record size will be interpreted as characters or words depending on *b*.
- b* is a letter (see description below for allowable letters) indicating whether the file is to contain formatted or unformatted records, or both.
- v* represents an unsubscripted integer variable called an associated variable. It receives a value after execution of its associated DEFINE FILE, READ, WRITE or FIND statement as described in the following paragraphs. The associated variable may not be used as an argument in a function or subroutine reference.

Description:

If the user has already assigned file u and it does not have enough space to contain m records, the OPEN statement may be used to extend the file. If the file is not assigned, then a dynamic assignment via @ASG,T (using m and s to calculate the file length) will be performed. The maximum number of records specification is only relevant when a direct access file is being defined for the first time.

If the record size, s , is in characters, the record will actually be stored in mass storage using word-sized locations. The number of words necessary to contain the record plus a constant and the record number are used to calculate the location in mass storage.

The b -field may be one of the following letters:

<u>Letter</u>	<u>Description</u>
L	File contains formatted or unformatted records or both. The maximum record size is measured in characters.
M	Same as L, except the file will be skeletonized if it is being created or extended.
E	File contains formatted records. The maximum record size is measured in characters.
F	Same as E, except the file will be skeletonized if it is being created or extended.
U	File contains unformatted records. The maximum record size is measured in number of words.
V	Same as U, except the file will be skeletonized if it is being created or extended.

At the conclusion of each READ and WRITE, v is set to a value that points to the record that immediately follows the last record transmitted. At the conclusion of a DEFINE FILE, v is set to one. At the conclusion of a FIND, v is set to a value that points to the record found. If the associated variable is used in a READ or WRITE statement to specify the record number, sequential processing is automatically secured. The associated variable cannot appear in the input/output list of a READ or WRITE statement for a file associated with the DEFINE FILE statement.

If more than one DEFINE FILE statement exists for a given file u , the first one encountered for the file is the one used during program execution. Subsequent descriptions of the file are ignored other than a validity check on the type (SDF, ANSI, symbiont) being done.

Examples:

```
DEFINE FILE 3(50,120,L,I), 4(120,50,L,J)
```

C This DEFINE FILE statement describes two files as numbers
C 3 and 4. The data in the first file consists of 50 records,
C each with a maximum length of 120 characters. The L
C specifies that the file may contain formatted and/or
C unformatted records. I is the associated variable that
C points to the next record.

C The data in the second file consists of 120 records,
C each with maximum length of 50 characters. The L
C specifies that the file contains formatted or
C unformatted records or both; J is the associated


```
C      variable that points to the next record.
      DEFINE FILE 3(50,30,U,I),4(120,13,U,J)
C      Same as the previous statement, except that the data is to
C      be transmitted without format control.

      DEFINE FILE 3(50,120,E,I),4(120,50,E,J)
C      Same as the first statement, except that a FORMAT
C      statement is required and the data is to be transmitted
C      under format control.
```

5.7.2. Direct Access READ Statement

Purpose:

The direct access READ statement causes records to be transferred from a file to internal storage. The file being read must have been previously defined with an OPEN or DEFINE FILE as a direct access file.

Form:

```
READ ([UNIT =] u [, [FMT =] f] ,REC = r [, ERR = s] [, IOSTAT = ios]) [iolist]
```

or:

```
READ (u'r [, f] [, ERR = s] [, IOSTAT = ios] ) [iolist]
```

where:

UNIT = is an optional clause (see 5.2.1).

u is the same file reference number (see 5.2.1) as for an associated OPEN or DEFINE FILE statement. The file number must be followed by an apostrophe for the second form.

r is an integer expression that represents the relative position of a record within the file (see 5.2.2). The REC= clause must be present for the first form. The REC=, UNIT=, and FMT= clauses must not be present for the second form of the READ.

FMT = is an optional clause (see 5.2.4).

f is a format specification (see 5.2.4).

ERR = *s* is an optional clause that will cause control to be transferred to statement label *s* if an error is detected in the execution of the READ statement (see 5.2.6).

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

iolist is an ordered input list of variables that are to receive the record read (see 5.2.3).

Description:

A record is read from position r of file u (which has been previously described by an OPEN or DEFINE FILE statement).

If the UNIT= clause is present for unformatted I/O statements, the specifiers may appear in any order. If the UNIT= clause is present for formatted I/O statements, the FMT= clause must be used.

The relative record number of the first record in a direct access file is one.

Example:

```
OPEN (UNIT=3, RECL=120, RCDS=50, RFORM='E', ASSOC=I)
```

```
READ(FMT=222, REC=43, UNIT=3, ERR=140) VAR1, ARR1
```

or:

```
READ (3'43, 222, ERR = 140) VAR1, ARR1
```

C These direct access reads transfer the contents of the
C 43rd record on unit 3 to VAR1 and array ARR1 according
C to the format specified at the statement labeled 222.

5.7.3. Direct Access WRITE Statement**Purpose:**

The direct access WRITE statement transfers records to mass storage from internal storage. The file being written must have been previously defined with an OPEN or DEFINE FILE statement as a direct access file.

Form:

```
WRITE ([UNIT =]  $u$  [, [FMT =]  $f$ ] , REC =  $r$  [, ERR =  $s$ ] [, IOSTAT =  $ios$ ]) [ $iolist$ ]
```

or:

```
WRITE ( $u$ ' $r$  [,  $f$ ] [, ERR =  $s$ ] [, IOSTAT =  $ios$ ] ) [ $iolist$ ]
```

where:

UNIT = is an optional clause (see 5.2.1).

u is the same file reference number (see 5.2.1) as for an associated OPEN or DEFINE FILE statement. The file number must be followed by an apostrophe for the second form.

r is an integer expression that represents the relative position of a record within the file (see 5.2.2). The REC= clause must be present for the first form. The REC=, UNIT=, and FMT= clauses must not be present in the second form of the WRITE.

- FMT = is an optional clause (see 5.2.4).
- f* is a format specification (see 5.2.4).
- ERR = *s* is an optional clause that will cause control to be transferred to statement label *s* if an error is detected in the execution of the WRITE statement (see 5.2.6).
- IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.
- iolist* is an ordered list of variables that are to be written (see 5.2.3).

Description:

A record is written to position *r* of file *u*. File *u* must have been previously described by an OPEN or DEFINE FILE statement. If the *iolist* of an unformatted direct access WRITE does not fill the record, the remainder of the record is undefined. If the *iolist* and the format of a formatted direct access WRITE do not fill a record, blank characters are added to fill the record.

If the UNIT= clause is present for unformatted I/O statements, the specifiers may appear in any order. If the UNIT= clause is present for formatted I/O statements, the FMT= clause must be used.

Example:

```
OPEN (UNIT=9, RECL=20, RCOS=75, RFORM='U', ASSOC=I)
```

```
WRITE(REC=53, ERR=140, UNIT=9)VAR1, ARR2
```

or:

```
WRITE (9, 53, ERR = 140) VAR1, ARR2
```

C These WRITE statements direct the contents of variable
C VAR1 and array ARR2 to be written into record 53
C of unit 9 without format control.

5.7.4. FIND Statement

Purpose:

The FIND statement prepositions a direct access file to a given relative position.

Form:

```
FIND ( [ UNIT = ] u, REC = r [ , ERR = s ] [ , IOSTAT = ios ] )
```

or:

```
FIND ( u' r [ , ERR = s ] [ , IOSTAT = ios ] )
```

where:

UNIT = is an optional clause (see 5.2.1).

u is a file reference number (see 5.2.1).

r is an integer expression specifying the relative position of a record within the file (see 5.2.2). The REC= clause must appear for the first form. The apostrophe must appear for the second form. The REC= and UNIT= must not appear in the second form of the FIND.

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

The FIND statement positions the direct access file to the record requested but returns control without causing any input/output operation to be completed. This may be used to locate the next input record while the present record is being processed, thereby increasing the execution speed of the object program.

Normally, records are written immediately following the previous record. They are written anywhere in a sector without regard to sector boundaries. As a result, it is advantageous to use the FIND statement prior to doing any output, if a particular alignment or positioning is required.

If the UNIT= clause appears, the specifiers may appear in any order for the second form.

Example:

```
      FIND (9'54)
C           Finds 54th record of output file 9.
```

```
      FIND (REC=44,UNIT=8)
C           Locates 44th record of input file 8.
```

5.8. Input/Output Contingencies

When an error occurs during ASCII FORTRAN I/O processing, the standard action is to print an error message, possibly call the ASCII FORTRAN Interactive Postmortem Dump (see 10.7), and terminate the program. To prevent program termination and to allow the use of files which contain multiple logical subfiles (for example, multiple file reels), the ERR, END, and IOSTAT contingency clauses can be used in certain input/output statements. When the ERR or END clause is used, the error message is not printed and control is passed to the statement specified by the particular contingency clause. In addition, the error status is encoded in an input/output status word which may be referenced using the functions IOC, IOS, and IOU. If the IOSTAT clause is present, control will be returned inline (without the error message being printed) unless an END or ERR clause is present. The input/output status is also returned via the IOSTAT clause.

5.8.1. Input/Output Contingency Clauses

The contingency clauses are written as optional parts of a particular input/output statement. The order of the clauses (if any are used) is immaterial. They are used to transfer control to the statement indicated if the specified contingency condition is encountered while executing the input/output statement. The forms of the contingency clauses are:

ERR = *s* (described in 5.2.6)

END = *sn* (described in 5.2.7)

IOSTAT = *ios* (described in 5.2.8)

When transfer is made to the statement specified by the ERR clause, the input/output status word will have been set to indicate the cause of the error or warning, the file reference number for the file in error, and, in certain cases, a substatus. See Appendix G for a detailed description of the input/output status word.

The three fields (cause, unit, substatus) of the I/O status word are all coded integers and may be tested or retrieved using the functions:

<u>Function</u>	<u>Field Reference</u>
IOC ()	cause
IOU ()	unit
IOS ()	substatus

The particular field referred to is set to zero following the reference. The I/O status word is set when an I/O error occurs; it remains set until the next I/O error occurs or until it is referred to through the functions IOC, IOU, and IOS, in which case only the field referred to is set to zero.

5.8.2. Input/Output Error Messages

There are two types of input/output errors:

- Those causing a SPERRY UNIVAC Series 1100 Executive contingency interrupt
- Those detected by the ASCII FORTRAN input/output modules

When a contingency interrupt occurs, control is given to the ASCII FORTRAN contingency routine. This routine prints the contingency error message, may call the the ASCII FORTRAN Interactive Postmortem Dump (see 10.7), closes all open files, and terminates the program. The ERR clause is not valid for this type of error.

When the error is detected by the I/O handler and no ERR or IOSTAT clause is specified, an error message is printed, all open files are closed, the FORTRAN Post Mortem Dump routine (FTNPMD) may be called, and the program is terminated. If the IOSTAT clause is present, the message is not printed and the program will continue inline. If the ERR clause is specified, the message is not printed but the I/O status word is set and transfer is made to the statement specified by the ERR clause.

Some error conditions are detected which are not considered serious enough to cause program termination. This less serious class of error conditions is referred to as warnings. If neither an ERR nor an IOSTAT clause is present when one of these conditions is detected, the I/O status word is set, a warning message is printed and execution of the current statement is continued. The presence of only an ERR clause results in the I/O status word being set, and transfer is made to the statement specified by the ERR clause without completing execution of the current statement.

The warning and error messages are itemized in Appendix G.

5.9. Internal Files

Internal files provide a means of transferring and converting data from internal storage to internal storage. The internal file READ and WRITE and the ENCODE and DECODE statements are similar to sequential access formatted READ and WRITE of external files. However, rather than transferring data between a peripheral file and mass storage, data is transferred between areas of main storage. Therefore, it is possible to move information from a storage block to a data list while manipulating it with format specifications without a physical peripheral device.

5.9.1. Internal File Formatted READ

Purpose:

The internal file formatted READ statement reads values into items specified in an input list according to a specified format.

Form:

```
READ ([UNIT =] u, [FMT =] f [, ERR = s] [, END = sn] [, IOSTAT = ios]) [iolist]
```

where:

- [UNIT =] *u* is a file reference specification. The variable *u* may be a character variable, character array, character array element or character substring. The UNIT= clause is an optional part of the specification.
- [FMT =] *f* is a format specification (see 5.2.4); *f* must be present but the FMT= clause is an optional part of the specification.
- ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.
- END = *sn* is an optional end clause specification (see 5.2.7); *sn* is a statement label.
- IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.
- iolist* is an input list (see 5.2.3).

Description:

The internal file formatted READ statement causes one or more formatted records to be read from the internal file specified. The file may be a character variable, character array, character array element or character substring.

A record of an internal file is a character variable, a character array element or a character substring. If the file is a character variable, character array element or a character substring, it consists of a single record whose length is the length of the character variable, array element or substring. If the file is a character array, each array element is a record of the file. The ordering of the records of the file is the ordering of the array elements in the array. The length of the record is the length of the array element.

The information in the record is scanned and converted as specified by the format specification *f*. The resulting values are assigned to variables specified in *iolist*. An empty FORMAT and an asterisk for *f* are not allowed for an internal file READ statement which contains *iolist*.

If the optional UNIT= and FMT= clauses appear, the control list specifications may appear in any order. If the UNIT= clause is present, the FMT= clause must appear.

An internal file is always positioned at the beginning of the first record before the READ.

The end-of-file condition will be raised if an attempt is made to read beyond the end of the internal file.

5.9.2. DECODE Statement

Purpose:

The DECODE statement converts characters in internal records starting at *b* to data items according to the format *f* and stores them in the I/O list items in *iolist*.

Form:

```
DECODE ( [ c, ] f, b [ , t ] [ , ERR = s ] ) iolist
```

where:

- | | |
|----------------|---|
| <i>c</i> | is an unsigned integer constant, integer parameter constant, integer variable or integer array element whose value specifies the number of characters per internal record to be scanned. |
| <i>f</i> | is the statement label of a FORMAT statement or the name of an integer variable containing the statement label of a FORMAT statement or the name of an array that contains format specifications. |
| <i>b</i> | is the name of an array (except assumed-size array), array element, or variable from which data is to be transferred. |
| <i>t</i> | is the name of an integer variable or array element which, upon completion of the operation, will contain the number of characters actually scanned during execution of the DECODE statement. |
| ERR = <i>s</i> | is an optional clause that will cause control to be transferred to statement label <i>s</i> if an error is detected in the execution of the DECODE statement (see 5.2.6). |
| <i>iolist</i> | is as specified for a READ statement and contains the items whose values are set by decoding the information in <i>b</i> . |

Description:

The fourth argument (*t*) and the fifth argument (ERR=*s*) may be omitted unconditionally. The first argument, *c*, with its following comma may be omitted if and only if the fourth argument is also omitted. If *c* is omitted, the internal record size is assumed to be 132 characters.

Execution of the DECODE statement causes the characters in the internal records in *b* to be converted to data items, according to the FORMAT specified by *f*, and stored into the elements specified in *iolist*. If the format is empty, the characters in the internal records are converted according to the type of items in the list. The characters of the internal records to be processed are contained in consecutive character positions of *b* with the first character of the first record in the first character position (leftmost quarter) of *b*. When a new record is to be processed, the first character of the record is contained in the first character position following the previous record. Thus the first character of the first record is contained in the first character position of *b*, the first character of the second record is contained in the (*c* + first) character position, etc. If *c* is omitted, the second record begins in character position 133.

If the interaction of *iolist* with *f* requires more characters to be read from a record than the count specified by *c*, or, if *c* is omitted, 132, a warning message is given. A list-directed DECODE can read more than one record.

Upon completion of execution of the DECODE statement, *r* if specified will contain the total number of characters scanned for the records processed. The count does not include a count of the trailing characters that are skipped over when terminating the processing of a record.

Example:

```
CHARACTER*7 P(6)
P(1)='00001.2'
P(2)='00023.0'
P(3)='00075.6'
2  FORMAT (3G7.2)
   DECODE (23,2,P,L) U,V,W
C      The DECODE statement would generate the following
C      internal values:
C          U = 1.2
C          V = 23.0
C          W = 75.6
C          L = 21
```

5.9.3. Internal File Formatted WRITE

Purpose:

The internal file formatted WRITE statement writes specified data into a specific internal file according to a given format.

Form:

```
WRITE ([UNIT =] u, [FMT =] f [,ERR = s] [,IOSTAT = ios]) [iolist]
```

where:

[UNIT =] *u* is a file reference specification (see 5.2.1). The variable *u* may be a character variable, character array, character array element or character substring. The UNIT= clause is an optional part of the specification.

[FMT =] *f* is a format specification (see 5.2.4); *f* must be present but the FMT= clause is an optional part of the specification.

ERR = *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

IOSTAT = *ios* is an optional I/O status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

iolist is an output list (see 5.2.3).

Description:

The internal file formatted WRITE statement causes one or more formatted records to be written to the internal file specified. The file may be a character variable, character array, character array element or character substring.

A record of an internal file is a character variable, character array element or a character substring. If the file is a character variable, array element or substring, it consists of a single record whose length is the length of the character variable, array element or substring. If the file is a character array, each array element is a record of the file. The ordering of the records in the file is the order of the array elements in the array. The length of the record in a character array is the length of the character array element.

The information in the list items are converted and stored as specified by the format specifications to the internal file. An empty format is not allowed. The asterisk may not be used for *f*.

If the optional UNIT= and FMT= clauses appear in the control list, the control list specifications may appear in any order. If the UNIT= clause is used, the FMT= clause must also be used.

An internal file is always positioned at the beginning of the first record before the WRITE.

5.9.4. ENCODE Statement

Purpose:

The ENCODE statement converts data items in *iolist* to character form according to the format *f* and places them into records in main storage starting at *b*.

Form:

```
ENCODE ([ c, ] f, b [ , t ] [ , ERR = s ] ) iolist
```

where:

- | | |
|----------------|---|
| <i>c</i> | is an unsigned integer constant, integer parameter constant, integer variable or integer array element whose value specifies the number of characters to generate per internal record. |
| <i>f</i> | is the statement label of a FORMAT statement or the name of an integer variable containing the statement label of a FORMAT statement or the name of an array that contains format specifications. |
| <i>b</i> | is the name of an array (except assumed-size array), array element, or a variable to which data is to be transferred. |
| <i>t</i> | is the name of an integer variable or array element which, upon completion of the operation, will contain the number of characters (excluding trailing blanks) actually generated by execution of the ENCODE statement. |
| ERR = <i>s</i> | is an optional clause that will cause control to be transferred to statement label <i>s</i> if an error is detected in the execution of the ENCODE statement. (See 5.2.6.) |
| <i>iolist</i> | is as specified for a WRITE statement and contains the items whose values are to be encoded and stored into <i>b</i> . |

Description:

The fourth argument (*t*) and the fifth argument (ERR = *s*) may be omitted unconditionally. The first argument, *c*, with the following comma may be omitted if and only if the fourth argument is also omitted.

If *c* is omitted, the internal record size in the block area is determined according to *f* and *iolist* with a maximum of 132 characters. However, the next internal record in the block area will start with the first character of the next word and the remainder (if any) of the last word will be blank filled. If an empty FORMAT statement is used, the internal record size will be 132 characters if *c* is omitted.

Execution of the ENCODE statement causes the data items specified by *iolist* to be converted to character strings according to the FORMAT specified by *f*. If an empty FORMAT is given, the data items in the list will be converted according to their type. The characters generated are placed in consecutive character positions of *b* with the first generated character going into the first character position (leftmost quarter) of *b*. When a new record is begun and *c* is specified, the first character generated is placed in the first character position following the previous record. Thus, the first character of the first record is placed in the first character position of *b*; the first character of the second record is placed in the (*c* + first character) position, and so forth.

If the interaction of *iolist* with *f* would cause more characters to be generated than specified by *c* (or 132 if *c* is not specified), the extra characters are lost and a warning message is given. They are not placed into the character positions following the current record. If the number of characters generated is less than that required to fill the record, the remainder of the record is filled with blanks. A list-directed ENCODE may write more than one record.

Upon completion of execution of the ENCODE statement, *r*, if specified, will contain a count of the total number of characters generated for the records. The count does not include any trailing blanks generated to fill out the records.

NOTE: *It is the user's responsibility to ensure that the area to which data is transferred (b) is large enough. If it is not, data transfer may not be complete or other areas may be overwritten unintentionally. The size c relates to the internal record size; the array b may receive multiple records.*

Example:

```
DIMENSION P(6)
R = 1.2
S = 23.0
T = 75.6
1  FORMAT (3G7.2)
   ENCODE (23,1,P,J) R,S,T
   END
```

C This will produce an array P which appears in storage
C as follows:

C 1.2ΔΔΔΔ23.ΔΔΔΔ76.ΔΔΔΔΔΔ

C Effectively, the numbers are rounded. The last
C two positions are blank-filled because the values
C stored are less than the record size. J will be
C equal to 21.

5.10. Auxiliary Input/Output Statements

There are three auxiliary input/output statements:

- OPEN (defines the characteristics of a particular external file)
- CLOSE (terminates the association of a particular external file to a particular unit and closes the file or changes the status of a unit from reread to closed)
- INQUIRE (returns information about the properties of a particular external file or the association to a particular unit)

The auxiliary input/output statements may be used with either sequential or direct access files. However, auxiliary input/output statements must not specify an internal file.

5.10.1. OPEN

Purpose:

An OPEN statement defines the characteristics of a particular file which may be used during a sequential or direct access input/output operation or which may be referenced by an auxiliary input/output (I/O) statement. The basic file types which can be defined are symbiont, ANSI, and sequential or direct SDF. In addition, the execution of the OPEN statement may:

- associate an existing file with a unit;
- create a file and associate it with a unit;
- change certain characteristics of an association between a file and a unit;
- create a file that is initially associated with a unit such as the symbionts or alternate symbionts;
- open the unit if the unit has not previously been opened with its associated file;
- assign a unit number for rereading the last formatted record read; or
- override and modify the symbiont code field in the FRT for a unit.

A unit may be opened by the execution of an OPEN statement in any program unit of an executable program and, once opened, may be referenced in any program unit of the executable program.

In the form which follows, the unit specifier is required to appear. The remaining clauses are optional, except the record length (RECL = *r/*), which must be provided if the file is being opened for direct access. The optional clauses are unordered. The unit specified must exist. Restrictions involving the combinations of clauses allowed are specified under the discussions of the individual clauses.

Form:

```

OPEN ([UNIT=] u [, IOSTAT= ios] [, ERR= s] [, FILE= fname] [, STATUS= sta]
      [, ACCESS= acc] [, FORM= fm] [, BLANK= blnk] [, RECL= rl]
      [, RFORM= rform] [, MRECL= mrc1] [, TYPE= type] [, BLOCK= bsize]
      [, OFF= boff] [, SEG= ssize] [, RCDS= mnum] [, ASSOC= var] [, REREAD= rrd]
      [, PREP= psize] [, BUFR= bufsize] )

```

where the clauses are:

[UNIT=] *u* *u* is an integer expression specifying a file reference number (see 5.2.1). If UNIT= is present, this clause need not be the first. If UNIT is absent, *u* must precede all other optional clauses. An asterisk may not be used for *u*.

IOSTAT= *ios* is an optional input/output status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

ERR= *s* is an optional error clause specification (see 5.2.5); *s* is a statement label.

FILE = *fname* *fname* is a character expression whose value is the name of the external file to be associated with the specified unit. Trailing blanks are ignored. The file name must be of the form [*qualifier**]*file name*[.]. Note that keys and F-cycles are not allowed. When this clause is present, a @USE *unit-num, qualifier * file name*. control statement will be executed if the unit number does not match the file name. This clause may not be present if a status of SCRATCH is specified. If this clause is omitted and the unit is not opened, the unit number will be used as the file name.

STATUS = *sta* *sta* is a character expression whose value is OLD, NEW, SCRATCH, EXTEND or UNKNOWN. Trailing blanks are ignored. If UNKNOWN is specified, a FILE= clause is optional. If OLD, NEW, or EXTEND is specified, a FILE= clause is optional.

If OLD is specified, the file must exist.

If NEW is specified and the file has not been preassigned to desired size, a file will be created (via an @ASG,CP statement) according to rules 3 and 4 given under the discussion on UNKNOWN. Note that new ASCII alternate symbiont files are not assigned by ASCII FORTRAN. These will be assigned by the Executive System in batch mode only.

If SCRATCH is specified, a file is created (via an @ASG,T statement) with a file name equal to the unit number. This file will be deleted at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program.

If EXTEND is specified, the file must exist.

For direct access files, EXTEND allows writing records up to a new maximum relative record count specified by the RCDS clause.

For sequential access files, **EXTEND** results in the file being positioned after the current last record of the file.

The following rules apply:

- If necessary, the file must be reassigned with a larger track size prior to execution.
- A file created on tape may only be extended while residing on tape.
- A file created on mass storage may be extended while residing on mass storage or tape; once the file is extended on tape it cannot be extended later when residing on mass storage.
- A tape file must be extended with the same block size used to create the file.
- A mass storage file which was created with a block size other than a multiple of 28 words must be extended with the same block size used to create the file.
- A mass storage file created with a block size equal to a multiple of 28 words can be extended with any block size which is a multiple of 28 words.
- For direct access files, the record format used when creating the file must also be used when extending the file. Thus if a file is skeletonized when initially created, skeletonization must also be specified when the file is extended.
- If the file is to be extended in the same executing program which created it, the file must have been closed prior to executing the **OPEN** statement which extends it.

If **UNKNOWN** is specified:

1. A check is made to see if the file is assigned to the run. If it is assigned, processing continues.
2. If the file is not assigned, an assignment via an **@ASG,A** statement is performed. If the file was cataloged and not rolled out, processing continues.
3. If the file is not cataloged and the access mode is direct, an assignment via a **@ASG,T file name , /// fsize** statement (**ASG,CP** if its status is **NEW**) is performed before processing continues. The *fsize* parameter is determined by:

$$\frac{mnum * (rl + nbr-control-words) + label-size + eof-word}{track-size}$$

where:

$$nbr\text{-}control\text{-}words = (rl + 2046)/2047$$
$$label\text{-}size = 112 \text{ words}$$
$$eof\text{-}word = 1 \text{ word}$$
$$track\text{-}size = 1792 \text{ words/track}$$
$$mnum = \text{see RCDS} = mnum \text{ clause}$$
$$rl = \text{record length in words, see RECL} = rl \text{ clause}$$

If $mnum$ has not been provided, $fsize$ will be given a value of 128 tracks.

4. If the file is not cataloged and the access mode is sequential, and an assignment via an @ASG,T *file-name* statement (@ASG,CP if the status is NEW) is performed before processing continues.

This discussion of UNKNOWN does not apply to ASCII symbiont files. Refer to Appendix G for a discussion on the handling of ASCII symbiont files.

Execution of a DEFINE FILE statement is equivalent to execution of an OPEN statement with a status of UNKNOWN and corresponding optional clauses specified.

If this clause is omitted, the default value is UNKNOWN.

ACCESS=*acc* *acc* is a character expression whose value is one of the following:

SEQUENTIAL

DIRECT

SEQ

DIR

Trailing blanks are ignored. This clause is used to specify the access method to be used with the file being opened.

If this clause is omitted, the default value is SEQUENTIAL.

FORM=*fm* *fm* is a character expression whose value is FORMATTED or UNFORMATTED. Trailing blanks are ignored. It specifies that the file is being opened for formatted or unformatted input/output, respectively.

For direct SDF, a value of FORMATTED implies a record format of E (for files containing formatted records, see the discussion on *rfm*) while a value of UNFORMATTED implies a record format of U (for files containing unformatted records, see the discussion on *rfm*).

If this optional clause is omitted and the RFORM optional clause is also absent, a value of UNFORMATTED is assumed if the file is being opened for direct access (a value of U is assumed for a record format; see the discussion on *rfm*). A value of FORMATTED is the default value if the file is being opened for sequential access with a file type other than ANSI. For ANSI files, the default value is UNKNOWN.

This optional clause may not appear if the RFORM clause is present and an SDF direct file is being opened.

RECL = *rl*

rl is an integer expression whose value must be positive. It specifies the length of each record in the file being opened for direct access. If the file is being opened for formatted input/output, the length is the number of characters. If the file is being opened for unformatted input/output, the length is measured in words.

This clause must appear if, and only if, a file is being opened for direct access.

BLANK = *blnk*

blnk is a character expression whose value is NULL or ZERO. Trailing blanks are ignored.

If NULL is specified, all blank characters in numeric 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.

This clause is only permitted for a file being connected for formatted input/output.

If this clause is omitted, the default value is NULL.

RFORM = *rfm*

rfm is a character expression whose value specifies the record format for sequential ANSI files or SDF direct files. Trailing blanks are ignored. Allowable values for sequential ANSI files are:

<u>Format</u>	<u>Description</u>
U	Undefined. The record is written with no control information appended to it. The records may be variable in length. Each data block contains only one record. If a record size is provided, it should indicate the maximum record size. This is the default value for ANSI files.
F	Fixed, unblocked. The record is written with no control information appended to it. All records are of the same length. Each data block contains only one record. If a record size is provided, it should indicate the exact record size.
FB	Fixed, blocked. Basically the same as fixed, unblocked; however, each block contains one or more records. Each data block has the same number of records except possibly the last block. The last block is truncated if it does not contain the full number of records. If a record size is provided, it should indicate the exact record size.

V	Variable, unblocked. The record is written with four characters of control information indicating the record length in ASCII characters appended to it. The records may be variable in length. The data block contains only one record. If a record size is provided, it should indicate the maximum record size, excluding four characters of record control.
VB	Variable, blocked. The same as variable, unblocked except that the block will contain as many complete records as possible.
VS	Variable, unblocked, segmented. The record is segmented at the block boundary. Appended to each record segment are five characters of control information. The first character is the record segment indicator and the remaining four are the segment length. The data block contains only one segment of the record. If a record size is provided, it should indicate the maximum record size excluding the five character record control segment.
VBS	Variable, blocked, segmented. Similar to variable, unblocked, segmented; however, each block may contain more than one record segment but never more than one segment of the same record.

The VBS and VS forms specify the ANSI S format. The V and VB forms specify the ANSI D format. The F and FB specify the ANSI F format. The U specifies the ANSI U format.

To read a pre-level 9R1 ANSI file (unpacked character data), append '66' to the appropriate record format. For instance, to read a pre-level 9R1 ANSI file which is fixed and blocked, the resulting clause would be RFORM = 'FB66'.

Allowable values for SDF direct files are:

<u>Format</u>	<u>Description</u>
L	The file contains formatted or unformatted records or both. The maximum record size is measured in characters.
M	Same as L, except the file will be skeletonized if it is being created or extended. Skeletonization consists of writing a dummy record for every record in a direct access file at the time the file is created. A dummy record just consists of a record control word with the record type field set to deleted.
E	The file contains formatted records. The maximum record size is measured in characters.
F	Same as E, except the file will be skeletonized if it is being created or extended.

U The file contains unformatted records. The maximum record size is measured in number of words. This is the default value if both the FORM and RFORM clauses are absent.

V Same as U, except the file will be skeletonized if it is being created or extended.

See Appendix G for a discussion on skeletonization.

This clause is not allowed if the FORM clause is present and an SDF direct file is being opened.

MRECL= *mrc* *mrc* is an integer expression whose value must be positive. It specifies the maximum record size for a file being connected for sequential access. The unit of specification is determined by the TYPE clause. The defaults are:

APRINT, APRNTA, AREADA	132 chars
APUNCH, APNCHA, AREAD	80 chars
SDF sequential	33 words
ANSI	132 chars

For symbionts:

APRINT, APRNTA, AREADA	$0 < mrc \leq 160$
AREAD	$0 < mrc \leq 132$
APUNCH, APNCHA	$0 < mrc \leq 80$

For SDF, maximum record size must be specified when records (other than unformatted records) larger than the default size are to be written. Maximum record size need not reflect unformatted records as they are read and written in segments.

For ANSI, the maximum record size must be specified when records larger than the default size are being read or written, regardless whether they are formatted or unformatted.

TYPE= *type* *type* is a character expression which specifies the file format for files being opened for sequential access. The possible values are:

SDF
ANSI
APRINT
APRNTA
APUNCH
APNCHA
AREAD
AREADA

The default value for this clause is SDF.

This clause may be used to override and modify the symbiont code field specified when the file reference table was built, provided the unit had never been opened before or had a file status of closed prior to the open. Refer to G.4 for processing of alternate symbiont files.

BLOCK= *bsize* *bsize* is an integer expression which specifies the block size for ANSI and SDF sequential files.

This clause is optional for SDF sequential files. If omitted, the default value is 224 words.

This clause is optional for ANSI files. If omitted, the default value is 132 characters.

The block size (*bsize*) for ANSI files is dependent on the maximum record size (*mrcl*), the buffer offset (*boff*), and the record format (*rfm*) as follows:

- For U or F record format:

$$bsize = mrcl + boff$$

- For V record format:

$$bsize = mrcl + 4 + boff$$

- For FB record format:

$$bsize = n (mrcl) + boff$$

- For a VB record format:

$$bsize \geq n (mrcl + 4) + boff$$

- For a VS or VBS record format:

The block size may be less than, equal to, or greater than the record size, but it must be large enough to contain any buffer offset and the five characters of control information appended to each record segment.

OFF= *boff* *boff* is an integer expression in the range 0 to 99 characters which specifies the ANSI buffer offset. The default value for this field is 0.

The variable *boff* must be specified if an ANSI input file contains buffer offset information in the front of the data blocks. This clause is not needed for ASCII FORTRAN produced ANSI tapes since no buffer offset information is appended to the data blocks.

SEG=ssize *ssize* is an integer expression which specifies the record segment size in words for sequential SDF files. The range is $1 \leq ssize \leq 2047$. Record segment size is the size that all records will be segmented at when they are written to the SDF file. It is also the size at which unformatted records will be processed as they are processed in segments. The default value for this clause is 111 words.

RCDS=mnum *mnum* is an integer expression which specifies the maximum number of records a direct access file will contain. This clause is only applicable when a direct access file is being opened for the first time or when a direct access file is being opened with STATUS='EXTEND'. If absent, the default value for *mnum* is determined as follows:

$$\frac{(size * track-size) - (label-size + eof-word)}{rl + nbr-control-words}$$

where:

size = 128 tracks (default file size if the file was not preassigned) or the preassigned size of the file.

track-size, *label-size*, *eof-word*, *nbr-control-words*—see the description under the STATUS=*sta* clause

rl = record length in words; see the RECL=*rl* clause

ASSOC=var *var* is an unsubscripted integer variable which is used as an associated variable. The associated variable receives the next relative record number for SDF direct files after the execution of a READ or WRITE statement involving the file, and the specified record number after the execution of a FIND statement. The associated variable will be initialized to a value of one by the OPEN statement. The associated variable may not be used as an argument in a function or subroutine reference.

This clause may only appear when the file is being opened for direct access.

REREAD=rrd *rrd* is a character expression whose value is ON. Trailing blanks are ignored.

The only other clauses which may be present when the REREAD clause is present are: ERR, UNIT and IOSTAT. The OPEN may not try to open a unit for REREAD while that unit is open as a normal I/O file. An implicit CLOSE will not be done.

The reread unit is used for rereading the last formatted SDF sequential or symbiont record read (see 5.6.1.5).

PREP=psize *psize* is an integer expression used to specify the physical prep factor of the device on which an SDF direct access file is stored.

This clause is optional for SDF direct files. If omitted, the default value is 112 words.

BUFR=*bufsize* *bufsize* is an integer expression which specifies the buffer size to be used for SDF direct files.

This clause is optional for SDF direct files. If omitted, the default buffer size is the minimum allowable buffer size given by the following formula:

$$\text{minimum } bufsize = rl + nbr\text{-control-words} + 2 * psize + 1$$

where:

rl = record length in words; see the RECL = *rl* clause

nbr-control-words = $(rl + 2046) / 2047$

psize = physical prepping factor; see the PREP = *psize* clause.

Description:

Specifiers which may be character expressions can be uppercase, lowercase, or a mixture of both.

Multiple OPEN statements involving the same file and unit number are permitted. Only the BLANK and MRECL (symbionts only) clauses may specify a value different from the one in effect.

If the file to be associated with the unit is not the same as the file which is currently associated with the unit, an implicit CLOSE statement will be executed for the unit. The default value for the STATUS = clause of the CLOSE statement is used when doing the implicit CLOSE. The unit will then be associated with the new file and the new file opened.

If a file is open, execution of an OPEN statement involving that file and a unit other than the unit currently associated with that file is not permitted. Note that an implicit CLOSE is not executed if the unit is currently a reread unit. An explicit CLOSE must be executed before the unit may be associated with a file.

Examples:

```
INTEGER OUTKOM
OPEN (10, IOSTAT=OUTKOM, ERR=50, STATUS='OLD')
```

This OPEN statement opens a file for sequential access.

If a file has not been associated with 10 via a @USE control statement, 10 will be used as the file name.

The status of OLD requires that the file must already exist, otherwise, an error condition will result.

Should an error condition be encountered while executing the OPEN statement, the variable OUTKOM will receive the contents of the I/O status word, PTIOE, contained in the storage control table, and control will be transferred to the statement with label 50.

The remaining applicable clauses and their default values are:

FORM = 'FORMATTED'

BLANK = 'NULL'

MRECL = 33

TYPE = 'SDF'

BLOCK = 224

SEG = 111

The next example is:

```
CHARACTER*20 FSTAT
FSTAT = 'UNKNOWN'
OPEN (12, FILE = 'DATA1', STATUS = FSTAT, ACCESS = 'DIRECT', RECL = 25, RCDS = 1000,
1    ASSOC = NREC)
```

This OPEN statement opens a file for direct access.

A @USE 12,DATA1 control statement will be executed. Since the status is UNKNOWN, an attempt will be made to assign the file if it is not yet assigned. If the file does not exist, a temporary file will be assigned.

Since neither the FORM nor the RFORM clause is present, the default record format applied specifies that the file will contain unformatted records. The file may contain up to 1000 records of 25 words each.

NREC may be interrogated to determine the next relative record number.

Should an error condition be encountered while executing the OPEN, an error message will be printed and the program will be terminated. Control will not be returned to the user as both the ERR and IOSTAT clauses are absent.

5.10.2. CLOSE

Purpose:

A CLOSE statement terminates the association of a particular external file with the specified unit and closes the file **or changes the status of a unit from reread to closed.**

Execution of a CLOSE statement for a particular 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 the unit.

In the form that follows, the UNIT clause is required to appear. The remaining clauses are optional and (if present) are unordered.

Form:

```
CLOSE ( [UNIT=] u [ , IOSTAT= ios ] [ , ERR= s ] [ , STATUS= sta ] [ [ , REREAD= rrd ] ] )
```

where:

[UNIT=]*u* *u* is an integer expression specifying a file reference number (see 5.2.1). If UNIT= is present, this clause need not be the first. If absent, *u* must precede all other optional clauses. An asterisk must not appear for *u*.

IOSTAT=*ios* is an optional input/output status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

ERR=*s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

STATUS=*sta* *sta* is a character expression whose value is one of the following:

KEEP

DELETE

REWIND

FREE

Trailing blanks are ignored. These values determine the disposition of the file that is being closed.

If KEEP is specified, the file continues to exist after execution of the CLOSE statement. KEEP must not be specified for a file whose OPEN status was specified as SCRATCH. A @FREE control statement is not executed if KEEP is specified unless the OPEN status was SCRATCH.

If DELETE is specified, the file ceases to exist after execution of the CLOSE statement. If the file was assigned with the C or U options, a @FREE,I control statement will be executed; otherwise, a @FREE,D control statement will be executed.

If REWIND is specified, the file is rewound and then treated as if a KEEP had been specified.

If FREE is specified, an @FREE control statement will be executed.

The default value is KEEP, unless the file's OPEN status was specified as SCRATCH, in which case the default value is DELETE.

REREAD *rrd* *rrd* is a character expression whose value is OFF. Trailing blanks are ignored.

The STATUS clause may not be present when the REREAD clause is present. The REREAD clause must not be present if the unit is open as a normal I/O file. This clause removes the assignment of the unit as a reread unit. After the CLOSE with REREAD, the unit may be associated with a normal I/O file within the same program unit (see 5.6.1.5).

Description:

Specifiers which may be character expressions can be uppercase, lowercase, or a combination of both.

Execution of a CLOSE statement specifying a unit that does not exist or has not been opened, affects no file and is permitted. Execution continues with the next statement.

After a unit is no longer associated with a file due to the execution of a CLOSE statement, it may be reassigned within the same executable program to the same file or a different file or it may be used as a reread unit.

After a file has been closed, it may be associated within the same executable program to the same unit or a different unit and opened again.

A CLOSE statement results in the file control buffers being freed for the file being closed. Thus, no position information on the file is available.

Example:

```
CLOSE (12,IOSTAT=MSTAT,ERR=100,STATUS='DELETE')
```

This CLOSE statement closes the file associated with unit 12.

Since DELETE was specified, the file will no longer exist after execution of the CLOSE statement.

Should an error condition be encountered while executing the CLOSE statement, the variable MSTAT will receive the contents of the I/O status word, PTIOE, contained in the storage control table, and control will be transferred to the statement with label 100.

After successful completion of this CLOSE, unit 12 will not be associated with any file.

5.10.3. INQUIRE

Purpose:

An INQUIRE statement returns information about the properties of a particular file or the association to a particular unit. There are two forms of the INQUIRE statement:

- INQUIRE by file The list of optional clauses must contain exactly one FILE clause and may not contain a UNIT clause or a REREAD clause.
- INQUIRE by unit The list of optional clauses must contain exactly one external UNIT clause and may not contain a FILE clause. The unit specified need not exist or be associated with a file.

The INQUIRE statement may be executed before, while, or after a file is opened. All values assigned by the INQUIRE statement are those that are current at the time the statement is executed.

The optional clauses of the INQUIRE statement which follows are unordered.

Form:

```
INQUIRE ( [UNIT=] u [ , FILE= fin ] [ , IOSTAT= ios ] [ , ERR= s ] [ , EXIST= ex ]
[ , OPENED= od ] [ , NUMBER= num ] [ , NAMED= nmd ] [ , NAME= fn ] [ , ACCESS= acc ]
[ , SEQUENTIAL= seq ] [ , DIRECT= dir ] [ , FORM= fm ] [ , FORMATTED= fmt ]
[ , UNFORMATTED= unf ] [ , RECL= rci ] [ , NEXTREC= nr ] [ , BLANK= blnk ]
[ , RFORM= rfm ] [ , MRECL= mrci ] [ , TYPE= typ ] [ , BLOCK= bsize ]
[ , OFF= boff ] [ , SEG= ssize ] [ , REREAD= rrd ] [ , PREP= psize ]
[ , BUFR= bufsize ] )
```

where:

[UNIT=]*u* *u* is an integer expression specifying a file reference number (see 5.2.1). If UNIT= is present, this clause need not be the first. If absent, *u* must precede all other optional clauses. An asterisk must not appear for *u*.

This clause must not be present if an INQUIRE by file is to be performed. The unit specified need not exist nor be associated with a file.

FILE= *fin* *fin* is a character expression whose value when any trailing blanks are removed is the name of the external file being inquired about. The file name must be of the form:

[*qualifier* *] *file-name* [.]

Keys and F-cycles may not be part of *fin* and will not be returned by the INQUIRE statement on *fin*.

This clause must not appear if an INQUIRE by unit is to be performed.

The named file need not exist nor be opened.

IOSTAT= *ios* is an optional input/output status clause specification (see 5.2.8); *ios* is an integer variable or integer array element.

ERR= *s* is an optional error clause specification (see 5.2.6); *s* is a statement label.

EXIST= *ex* *ex* is a logical variable or logical array element. Execution of an INQUIRE by file statement causes *ex* to be assigned the value .TRUE. if there exists a file with the specified name; otherwise *ex* is assigned the value .FALSE.

A file is said to exist if it meets at least one of the following conditions:

1. The file is associated with an opened unit.
2. The file is assigned to the run.
3. The file is already cataloged and can be assigned to the run.

Execution of an INQUIRE by unit statement causes *ex* to be assigned the value .TRUE. if the specified unit exists; otherwise, *ex* is assigned the value .FALSE.

A unit is said to exist if:

$$0 \leq \text{unit number} \leq \text{FRT length}$$

where FRT is the file reference table (see G.5).

OPENED= *od* *od* is a logical variable or logical array element. Execution of an INQUIRE by file statement causes *od* to be assigned the value .TRUE. if the file specified is opened; otherwise, *od* is assigned the value .FALSE.

Execution of an INQUIRE by unit statement causes *od* to be assigned the value .TRUE. if the specified unit is open; otherwise, *od* is assigned the value .FALSE.

NUMBER= *num* *num* is an integer variable or integer array element that is to receive the value of the external unit identifier if the unit is open. For an INQUIRE by file, the unit in question is the unit associated with the file. If no unit is associated with the file, *num* becomes undefined.

NAMED= *nmd* *nmd* is a logical variable or logical array element that is assigned a .TRUE. or .FALSE. value which indicates the existence of an external file name based on the following conditions.

For an INQUIRE by file, *nmd* is assigned a value of .TRUE. if the file exists; it need not be opened.

For an INQUIRE by unit, *nmd* is assigned a value of .TRUE. if the unit is opened and not associated with a symbiont. A value of .FALSE. is returned if the unit is opened and associated with a symbiont.

NAME= *fn* *fn* is a character variable or character array element that is assigned the value of the external name of the file if the file has a external name; otherwise, it becomes undefined.

The value returned by this optional clause is of the form:

*qualifier*file-name .*

ACCESS= *acc* *acc* is a character variable or character array element that is assigned the value SEQUENTIAL if the file is opened for sequential access, and DIRECT if the file is opened for direct access. If the file is not opened, *acc* becomes undefined.

SEQUENTIAL= *seq* *seq* is a character variable or character array element that is assigned the value YES if the file is opened for sequential access and a value of NO if the file is not opened for sequential access. A value of UNKNOWN is returned if the access method for the file cannot be determined.

If the file is not open (INQUIRE by unit) or if the file does not exist (INQUIRE by file), *seq* becomes undefined.

DIRECT= *dir* *dir* is a character variable or character array element that is assigned the value YES if the file is opened for direct access and a value of NO if the file is not opened for direct access. A value of UNKNOWN is returned if the access method for the file cannot be determined.

If the file is not open (INQUIRE by unit) or if the file does not exist (INQUIRE by file), *dir* becomes undefined.

FORM= *fm* *fm* is a character variable or character array element that is assigned the value FORMATTED if the file is opened for formatted I/O, UNFORMATTED if the file is opened for unformatted I/O, and BOTH if the I/O type is mixed. If the file is not open, *fm* becomes undefined. List-directed and namelist I/O are considered formatted.

FORMATTED= *fmt* *fmt* is a character variable or character array element that is assigned the value YES if the I/O type for the file is formatted, and NO if the I/O type for the file is not formatted. A value of UNKNOWN is returned if the I/O type of the file cannot be determined.

If the file is not open (INQUIRE by unit) or if the file does not exist (INQUIRE by file), *fmt* becomes undefined.

**UNFORMATTED
= *unf*** *unf* is a character variable or character array element that is assigned the value YES if the I/O type for the file is unformatted, and NO if the I/O type for the file is not unformatted. A value of UNKNOWN is returned if the I/O type of the file cannot be determined.

If the file is not open (INQUIRE by unit) or if the file does not exist (INQUIRE by file), *unf* becomes undefined.

RECL = *rci*

rci is an integer variable or integer array element that is assigned the value of the record length of the file if it is opened for SDF direct access.

If the file is opened for formatted input/output or can contain both formatted and unformatted records, the length is in characters. The length is measured in words if the file is opened for unformatted input/output.

If the file is not opened or if the file is not opened for direct access, *rci* becomes undefined.

NEXTREC = *nr*

nr is an integer variable or integer array element which receives the next relative record number for the file if it is opened for SDF direct access. If the file is opened but no records have been read or written since the opening, *nr* is assigned the value one.

If the file is not opened for direct access or if the position of the file is indeterminate due to a previous error condition, *nr* becomes undefined.

BLANK = *blnk*

blnk is a character variable or character array element. A value of NULL is returned if the file is opened for formatted input/output and null blank control is in effect. A value of ZERO is returned if the file is opened for formatted input/output and if zero blank control is in effect.

If the file is not opened or if it is not opened for formatted input/output, *blnk* becomes undefined.

RFORM = *rfm*

rfm is a character variable or character array element which will receive a literal value representing the record format for ANSI files and SDF direct files.

If the file is opened as an ANSI file, one of the following values will be returned:

<u>Value Returned</u>	<u>Record Format</u>
U	Undefined
F	Fixed, unblocked
FB	Fixed, blocked
V	Variable, unblocked
VB	Variable, blocked
VS	Variable, unblocked, segmented
VBS	Variable, blocked, segmented

If the file is opened as an SDF direct file, one of the following values will be returned:

<u>Value Returned</u>	<u>Record Format</u>
L	Formatted or unformatted records or both. The maximum record size is measured in characters.
M	Same as L except the file has been skeletonized.
E	Formatted records. The maximum record size is measured in characters.
F	Same as E, except the file has been skeletonized.
U	Unformatted records. The maximum record size is measured in words.
V	Same as U, except the file has been skeletonized.

The *rfm* variable becomes undefined for all other cases.

MRECL= *mrcf*

mrcf is an integer variable or integer array element that is assigned the value of the maximum record length of the file if it is opened for sequential access.

If the file is not opened or if the file is not opened for sequential access, *mrcf* becomes undefined.

The applicable units of measure are:

symblonts	characters
SDF sequential	words
ANSI	characters

TYPE= *typ*

typ is a character variable or character array element that is assigned one of the following file format values if the file is opened for sequential access:

SDF
ANSI
AREAD
AREADA
APUNCH
APNCHA
APRINT
APRNTA

If the file is opened for SDF direct or not open, *typ* becomes undefined.

BLOCK= <i>bsize</i>	<p><i>bsize</i> is an integer variable or integer array element which receives the block size if the file is opened with a file format of ANSI or sequential SDF.</p> <p>For SDF sequential files, the unit of measure is words.</p> <p>For ANSI files, the unit of measure is characters.</p> <p>For any other file format or if the file is not open, <i>bsize</i> becomes undefined.</p>
OFF= <i>boff</i>	<p><i>boff</i> is an integer variable or integer array element which receives the ANSI buffer offset in characters if the file is opened with a file format of ANSI.</p> <p>For any other file format or if the file is not opened, <i>boff</i> becomes undefined.</p>
SEG= <i>ssize</i>	<p><i>ssize</i> is an integer variable or integer array element that is assigned the value of the SDF record segment size in words. This value is defined only for files opened for SDF sequential access.</p>
REREAD= <i>rrd</i>	<p><i>rrd</i> is a character variable or character array element that is assigned the value ON if the unit exists and is assigned as a REREAD unit, and a value OFF if the unit exists and is not assigned as a REREAD unit. If the unit does not exist, <i>rrd</i> becomes undefined.</p> <p>All other clauses except the UNIT, IOSTAT, ERR, and EXIST clauses are ignored if the unit is a REREAD unit (see 5.6.1.5).</p>
PREP= <i>psize</i>	<p><i>psize</i> is an integer variable or integer array element which receives the physical prep factor specified for the device on which an SDF direct access file is stored.</p> <p>If the file is not opened for direct access, <i>psize</i> becomes undefined.</p>
BUFR= <i>bufsize</i>	<p><i>bufsize</i> is an integer variable or integer array element which receives the buffer size used for an SDF direct access file.</p> <p>If the file is not opened for direct access, <i>bufsize</i> becomes undefined.</p>

Description:

A variable or array element that receives a value in an INQUIRE statement may not be referred to by more than one of the clauses in the same INQUIRE statement.

For the execution of an INQUIRE by file statement:

- The variables *nmd*, *fn*, *seq*, *dir*, *fmt*, and *unf* are assigned values only if the value of *fin* is acceptable and if a file by that name exists; otherwise, they become undefined.
- The variable *num* becomes defined if and only if *od* becomes defined with a value of .TRUE. .
- The variables *acc*, *fm*, *rcl*, *nr*, *blnk*, *rfm*, *mrcl*, *typ*, *bsize*, *boff*, *psize*, *bufsize*, and *ssize* may become defined only if *od* becomes defined with a value of .TRUE. .

Execution of an INQUIRE by unit statement causes *num*, *nmd*, *fn*, *acc*, *seq*, *dir*, *fm*, *fmt*, *unf*, *rcl*, *nr*, *blnk*, *rfm*, *mrcl*, *typ*, *bsize*, *boff*, *psize*, *bufsize*, and *ssize* to be assigned values only if the specified unit exists and is opened; otherwise, they become undefined.

If an error condition occurs during execution of an INQUIRE statement, all of the optional clause variables and array elements except *ios* become undefined.

Variables *ex* and *od* always become defined unless an error condition occurs.

If the receiving area for a returned literal value is too small, a warning will be issued and the literal will be truncated on the right. Character literal values returned by INQUIRE will be uppercase.

Example:

```
CHARACTER*20  FN1,ACC1,SEQ1,DIR1, FORM1,FMT1,UNF1,RFORM1,
1BLANK1,TYPE1
  INTEGER  OUTKOM,UNIT1,RECL1,ASSOC,BLOCK1,BOFF1,SEG1
  LOGICAL  EXIST1,OPEN1,NAMED1
  INQUIRE
  (10,IOSTAT=OUTKOM,ERR=50,EXIST=EXIST1,OPENED=OPEN1,NUMBER=UNIT1,
1NAMED=NAMED1,NAME=FN1,ACCESS=ACC1,SEQUENTIAL=SEQ1,DIRECT=DIR1,FORM=FORM1,
2FORMATTED=FMT1,UNFORMATTED=UNF1,RECL=RECL1,NEXTREC=ASSOC,BLANK=BLANK1,
3RFORM=RFORM1,MRECL=MRECL1,TYPE=TYPE1,BLOCK=BLOCK1,OFF=BOFF1,SEG=SEG1)
```

Should an error condition be encountered while executing the INQUIRE, the variable OUTKOM will receive the contents of the I/O status word, PTIOE, contained in the storage control table, and control will be transferred to the statement with label 50.

Assuming this INQUIRE statement was executed prior to the OPEN statement in the first example in 5.10.1 and that unit 10 had not been opened otherwise, the following values would be returned:

```
OUTKOM = 0
EXIST1 = .TRUE.
OPEN1 = .FALSE.
```

Since that unit is not open, all other variables would be undefined.

Assuming the INQUIRE statement was executed immediately after the OPEN statement referred to previously, the following values would be returned:

OUTKOM = 0	
EXIST1 = .TRUE.	
OPEN1 = .TRUE.	
UNIT1 = 10	
NAMED1 = .TRUE.	
FN1 = 'TESTING*10.'	
ACC1 = 'SEQUENTIAL'	
SEQ1 = 'YES'	
DIR1 = 'NO'	
FORM1 = 'FORMATTED'	
FMT1 = 'YES'	
UNF1 = 'NO'	
RECL1 = undefined	<i>not applicable for SDF sequential</i>
ASSOC = undefined	<i>not applicable for SDF sequential</i>
BLANK1 = 'NULL'	
RFORM1 = undefined	<i>not applicable for SDF sequential</i>
MRECL1 = 33	
TYPE1 = 'SDF'	
BLOCK1 = 224	
BOFF1 = undefined	<i>not applicable for SDF sequential</i>
SEG1 = 111	

6. Specification and Data Assignment Statements

6.1. Overview of Specification Statements

Specification statements describe the data used in a program unit. The description may specify the following information:

- Data type (which in turn specifies the storage required for the data and its internal representation)
- Initial value
- Array declaration
- Data which is to share storage within a program unit
- Data which is to share storage among several program units

The specification statements are: DIMENSION, IMPLICIT, Explicit Typing, EQUIVALENCE, COMMON, BANK, PARAMETER, EXTERNAL, INTRINSIC, and SAVE. The EXTERNAL, INTRINSIC, and SAVE statements are explained in Section 7.

In an internal subprogram, names used in Explicit Typing, DIMENSION, COMMON, NAMELIST, PARAMETER, EXTERNAL, INTRINSIC, SAVE, and DEFINE statements will be local to the internal subprogram, even if the names are also used in its external program unit. See 7.1.1 for a description of global and local names.

Specification statements are nonexecutable. They may appear at the beginning of the program unit prior to the first executable statement, or they may appear following an ENTRY statement with no intervening executable statements. If specification statements follow an ENTRY statement, the variables described must not appear in a prior executable statement. If a variable appears in a DATA statement, any specification statements which describe the variable must precede the DATA statement. To conform to FORTRAN 77, DATA statements should follow all specification statements. DATA statements and statement ordering are discussed further in 6.8.1.

The EXTERNAL and INTRINSIC statements are discussed in 7.2.3 and 7.2.4, respectively. The SAVE statement is discussed in 7.12. The DATA statement is discussed in 6.8.1. BLOCK DATA is discussed in 7.8.

6.2. DIMENSION Statement

Purpose:

The DIMENSION statement declares arrays and specifies dimension information.

Form:

```
DIMENSION a(d [, d ]... ) [ /x/ ] [ , a(d [, d ]... ) [ /x/ ] ] ..
```

where:

a is a symbolic name.

d is a dimension declarator and specifies the extent of *a*; *d* must satisfy the requirements for dimension declarators (see 2.2.2.4.1).

x is initialization information for elements of *a*. It must satisfy the requirements of a constant list as specified in 6.8.

Description:

The appearance of a symbolic name in a DIMENSION statement declares that name as an array name. A symbolic name may also be declared as an array name in a COMMON statement or an explicit type statement. However, only one array declaration for a symbolic name is permitted in a program unit. The maximum number of elements an array can have is 262,143. An array element whose relative address under its location counter in the relocatable element is over 65536 cannot be initialized via a constant list or DATA statement.

If an array *a* has adjustable or assumed-size dimensions, it can be declared only in a subprogram and must be a dummy argument.

For a description of local-global rules for names in DIMENSION statements in internal subprograms, see 7.11.

Examples:

```
DIMENSION ARR1(5,5,2)
C      Symbolic name ARR1 is declared as a three-dimensional
C      array name. The maximum values that the three
C      subscripts may assume are 5, 5, and 2
C      respectively. The lower bounds for the three
C      subscripts are assumed to be one.

DIMENSION FACTOR (0:4) /1,2,3,4,5/
C      Symbolic name FACTOR is declared as a one-dimensional
C      array with five elements. A lower dimension bound
C      of zero and an upper dimension bound of four are
C      specified. The five elements are assigned initial
C      values 1 through 5 respectively.

SUBROUTINE ADJUST(ONE,THIRD,YES,ARR4)
INTEGER ONE,THIRD
DIMENSION ARR4(ONE,2,THIRD)
C      Symbolic name ARR4 is an adjustable dummy array.
C      The addresses of its contents will be calculated
C      using the variable values. No storage is actually
C      assigned to the array since it is a dummy array.

SUBROUTINE DETER(ARR5,ARR6, IDIM)
DIMENSION ARR5(-IDIM:2+IDIM, -IDIM:*), ARR6(*)
C      ARR5 and ARR6 are assumed-size dummy arrays.
C      ARR5 is a double-dimensioned array and ARR6 is
C      a single-dimensioned array. The value of a
C      subscript expression in a dummy element reference
C      must be in the range determined by the lower bound and
C      upper bound expressions specified for that dimension.
```

6.3. Type Statements

The type statements are used to specifically declare the characteristics of items. Otherwise, the normal convention specifies REAL type for any data item whose symbolic name begins with any of the letters A through H or O through Z, and INTEGER type for any data item whose symbolic name begins with any of the letters I through N.

Any data type from Table 6-1 may be used in a type statement.

Table 6-1. Valid Data Type and Length

Type	Permitted Length Specification	
	Default	Alternate
INTEGER	4	None
REAL	4	8
COMPLEX	8	16
LOGICAL	4	None
CHARACTER	1	2, 3, 4, . . . , 511 or *
DOUBLE PRECISION	8	None

Type statements may be either implicit or explicit.

Typical type and length specifications which could be used in type statements are INTEGER, LOGICAL, REAL (or REAL*4) for type single precision real, DOUBLE PRECISION (or REAL*8) for type double precision real, COMPLEX (or COMPLEX*8) for type single precision complex, COMPLEX*16 for type double precision complex, and CHARACTER*n (where n is a positive unsigned integer constant) for type character.

6.3.1. IMPLICIT Statement

Purpose:

The IMPLICIT statement assigns a specific data type to a symbolic name based on the initial alphabetic character of its name.

Form:

```
IMPLICIT t[*i] ( a [ , a ] ... ) [ , t[*i] ( a [ , a ] ... ) ] ...
```

where:

- t* is a data type chosen from Table 6-1.
- i* is an unsigned integer constant (or positive integer constant expression enclosed in parenthesis if the type is character) and is a permitted length specification for the given type. If the length is omitted, the standard length for the given type is assumed (see Table 6-1). Note that FORTRAN 77 allows **i* for type character only. **ASCII FORTRAN allows **i* for all data types.**
- a* is a single alphabetic letter or a range of alphabetic letters. A range is specified by the first and last letters of the range separated by a hyphen. For example, the notation B-F specifies letters B, C, D, E, and F and has the same effect as if all letters were listed separately.

Description:

The IMPLICIT statement affects the letter association used to assign data types by the name rule (see 2.2.2.2.1).

All letters, *a*, included in a parenthesized list are associated with the data type *t* and length *i* preceding that list. Within the program unit, all variables, arrays, parameter constants, function subprograms, and statement functions whose symbolic name begins with the letter *a* is given the associated data type if the name is not explicitly typed. IMPLICIT statements do not change the type of any intrinsic functions.

In a subprogram, IMPLICIT statements must appear after the SUBROUTINE, FUNCTION, or BLOCK DATA statement.

A program unit may contain any number of IMPLICIT statements, but IMPLICIT statements should precede all other specification statements, except PARAMETER statements. **ASCII FORTRAN allows DATA statements to precede IMPLICIT statements.**

If an IMPLICIT statement occurs after other specification statements or after an executable statement, a warning is issued and only statements following the IMPLICIT statement are affected by it.

A letter may neither appear in, nor be included in, a range specification of more than one IMPLICIT statement. Furthermore, a letter may appear only once in any IMPLICIT statement.

An internal subprogram gets the same IMPLICIT type associations as its external program unit. However, it may also have its own IMPLICIT statement which only affects its local names. Any following internal subprograms would have their IMPLICIT types revert to that of their external program unit (see 7.4.2 and 7.4.3).

Examples:

```
IMPLICIT LOGICAL (B)
C           Variables whose names begin with the letter 'B' are given
C           the LOGICAL type unless explicitly typed otherwise.
```

IMPLICIT CHARACTER*4(C-E,Z), COMPLEX*16(M-O)

C Variables whose names begin with the letters 'C', 'D', 'E',
C or 'Z' are given the CHARACTER type and a length of 4.
C Data whose names begin with letters 'M', 'N', or 'O' are
C given the COMPLEX type and their real and imaginary
C parts are double precision.

IMPLICIT LOGICAL (L)**IMPLICIT REAL (L)**

C Untyped variables having the initial letter 'L' which
C appear for the first time after the first IMPLICIT
C statement will be type logical. Untyped variables starting
C with the letter 'L' which appear for the first time
C following the second IMPLICIT will have type real.
C A compiler warning is issued when the second appearance
C of letter 'L' in an IMPLICIT statement in the same program
C unit is detected.

6.3.2. Explicit Type Statements

Purpose:

Explicit type statements assign a particular data type to a data item based on its name rather than its initial letter. In addition to data type, the following can be specified: dimension information, data length, and initial values.

If an initial value list is present, it applies to the immediately preceding array or variable only. Note that the syntax for initial values is different than in the DATA statement. Here, each data item has its own initial value list; whereas, in the DATA statement the values for many variables may be in one list.

A function name, parameter constant name, statement function name, or dummy function name may appear in an explicit type statement but may not be assigned an initial value.

If no data length is specified, the standard data length for that type is assumed. See Table 6-1.

If any intrinsic function name appears in an explicit type statement and the statement type is not the same as the predefined intrinsic type, the name becomes a user-defined name. If the types are the same, the definition is not changed.

The FORTRAN 77 standard states that the name of an intrinsic function must appear in an EXTERNAL statement to be able to define a user-defined function name. ASCII FORTRAN does not require the occurrence of the intrinsic function name in an EXTERNAL statement to define a user-defined function name.

6.3.2.1. INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL Type Statements

Form:

 $typ \ [*len] \ name \ [/x/] \ [, \ name \ [/x/] \] \dots$

where:

 typ is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL. len is an unsigned integer constant and is one of the permitted specifications for the given type typ (see Table 6-1). The length specified with type typ applies to all symbolic names in the statement except those names which are followed by a length specification. $name$ is one of the forms: $v \ [*len]$
 $ary \ [*len] [(d)]$

where:

 v is a variable name, parameter constant name, function name, statement-function name, or dummy function name. ary is an array name. d is dimension information which must satisfy the requirements of 2.2.2.4. x is initialization information which must satisfy the requirements of a constant list as specified in 6.8.1.

Examples:

```
INTEGER    SUM, ITEM(10)
*          SUM is specified as type integer and length 4.
*          ITEM is a one-dimensional array with 10 elements.
*          Each element of the array is type integer with length 4.
```

```
COMPLEX    C1, C2*16, C3
*          C1, C2 and C3 are specified as type complex.
*          C1 and C3 have a length of 8, C2 has length 16.
```

```
DOUBLE PRECISION  ARR(2) / 2*1.0 D+0 /
*          ARR is specified as a type real, single
*          dimensional array with two elements. Each
*          element is type real with length 8 or double
*          precision. Both elements of the array are
*          initialized with the double precision constant 1.0 D+0.
```

6.3.2.2. CHARACTER Type Statement

Form:

CHARACTER [** len*[,]] *name* [/x/] [, *name* [/x/]] . . .

where:

len is an unsigned integer constant or integer constant expression enclosed in parenthesis with a positive value in the range 1 to 511. It may also be an asterisk in parenthesis, (*). The length specified with CHARACTER applies to all symbolic names which are followed by a length specification.

name is one of the forms:

```
v[* len]  
ary[ (d) ] [* len]  
ary[* len][ (d) ]
```

where:

v is a variable name, parameter constant name, function name, statement function name, or dummy function name.

ary is an array name.

d is dimension information which must satisfy the requirements of 2.2.2.4.

x is initialization information which must satisfy the requirements of a constant list as specified in 6.8.1.

Description:

An entity declared in a CHARACTER statement may have an asterisk as its length specification if the entity is a function subprogram, a dummy argument, a parameter constant name, or a statement function name.

- If a dummy argument has an asterisk for *i*, the dummy argument assumes the length of the associated actual argument. If the associated actual argument is an array name, the length assumed by the dummy argument is the length of an array element in the associated actual argument array.
- If a function subprogram has an asterisk for *i*, the function name must appear in a FUNCTION or ENTRY statement in the same subprogram. When a reference to such a function is executed, the function assumes the length specified in the referencing program unit. The length specified for a character function in a program unit that refers to the function must be an integer constant expression and must agree with the length specified in the subprogram that specifies the function. Note that there is always agreement of length if an asterisk for *i* is specified in the subprogram that specifies the function.

- If a parameter constant name has an asterisk for *i*, the constant assumes the length of its corresponding constant expression in a PARAMETER statement.
- If a statement function name has an asterisk for *i*, then the expression generated by a statement function reference will be left as is, with no length conversion (see 7.4.1.2).

Examples:

```
CHARACTER*4    CHARA, CHARB(10)*1, CHARC
*             Variables CHARA and CHARC are specified with
*             type character and length 4. CHARB is a 1-
*             dimension array with 10 elements. Each element
*             of the array has type character and a length of 1.
```

```
CHARACTER      TITLE(2)*12 / 'PROGRAMMER', 'ANALYST' /
*             TITLE is specified as a type character, 1-
*             dimensional array with two elements. Each
*             character type element has length 12. The
*             first element is initialized with the character
*             constant 'PROGRAMMER' and the second is
*             initialized with the character constant 'ANALYST'.
```

6.4. EQUIVALENCE Statement

Purpose:

The EQUIVALENCE statement specifies sharing of storage by two or more data entities within a program unit.

Form:

EQUIVALENCE ($n, n[, n] \dots$) [, ($n, n[, n] \dots$)] \dots

where n is a variable name, an array element name, an array name, or a character substring name; n may not be a dummy argument name or function name. At least two names must appear in each parenthesized list.

Description:

All entities in a given parenthesized list share some or all of the same storage locations. The order of items in the list is not important.

Although variables of different types may be equivalenced, neither mathematical equality nor type conversion of these entities is implied. Equivalencing entities of different classes (an array and a scalar, for example) does not cause the entities to assume the same class. Rather the EQUIVALENCE statement permits the user to reduce the storage requirements of a program unit by causing two or more data entities to share the same storage locations. However, it is the user's responsibility to ensure that the logic of a program unit permits such storage sharing and that sharing does not destroy needed information.

A substring expression in an EQUIVALENCE list must be an integer constant expression.

Character entities may be equivalenced to one another. The length of the equivalenced entities are not required to be the same.

Character entities may be equivalenced to noncharacter type entities if the equivalence does not force the noncharacter entities to begin on nonword boundaries. See the example at the end of this section.

If an array name appears in an EQUIVALENCE statement, it may be followed by a parenthesized list of subscripts. Each subscript must be an integer constant expression and when evaluated may be positive, negative or zero. Such a list specifies a particular array element.

Use of an array name without a subscript list in an EQUIVALENCE list has the same meaning as using a subscript list which specifies the first array element. A subscript list may contain either one subscript or d subscripts, where d is the number of dimensions in the array declaration for that particular array. If the list has d subscripts, it refers to the array element in the usual way. If the list has one subscript, it refers to the linear position in the array starting at one (that is, the first element of the array is referenced by a subscript value of 1). For example, if array ARR1 has the dimensions (3,3), the equivalence reference ARR1(8) would refer to element ARR1(2,3). A subscript list with one subscript creates an ambiguity for single dimensioned arrays with a lower dimension bound other than one. For this case, the subscript refers to the array element in the usual manner rather than the linear position. For example, if ARR2 has the dimensions (-3:3), the equivalence reference ARR2(1) refers to the fifth element and not the first.

Equivalencing elements in two different arrays may implicitly equivalence other elements of these arrays.

If an entity appears in more than one list, the implication is that all elements of both lists share the same storage and these lists are combined into one single list.

Sharing is accomplished on the basis of storage words. The number of words needed for internal representation of the various data types is listed in Table 6-2.

Table 6-2. Storage Units Required for Data Storage

Data Type	Words
INTEGER	1
REAL	1
DOUBLE PRECISION	2
COMPLEX	2
COMPLEX*16	4
LOGICAL	1
CHARACTER*n	(see 6.9.1)

Examples:

```
REAL AX(10,10), BX, T, Z
INTEGER L
```

```
EQUIVALENCE (AX,Z), (BX,L,T)
```

C Variables AX and Z share the same storage locations,
C as do entities BX, L and T.

```
REAL AA(10,10), BB, Z, CC(10)
INTEGER M(20)
```

```
EQUIVALENCE (AA(3,3),BB,CC(1)), (BB,Z,M(1))
```

C Array element AA(3,3), variable BB, and array element CC(1)
C share the same storage location. BB, Z, and M(1)
C share the same location. Because entity BB appears in both
C lists, the lists are combined so that AA(3,3), BB, CC(1),
C Z, and M(1) all share the same storage location.

```
DIMENSION ARR1(8), KARR(3,2)
```

```
COMPLEX COMPA(1,2,2), CA
```

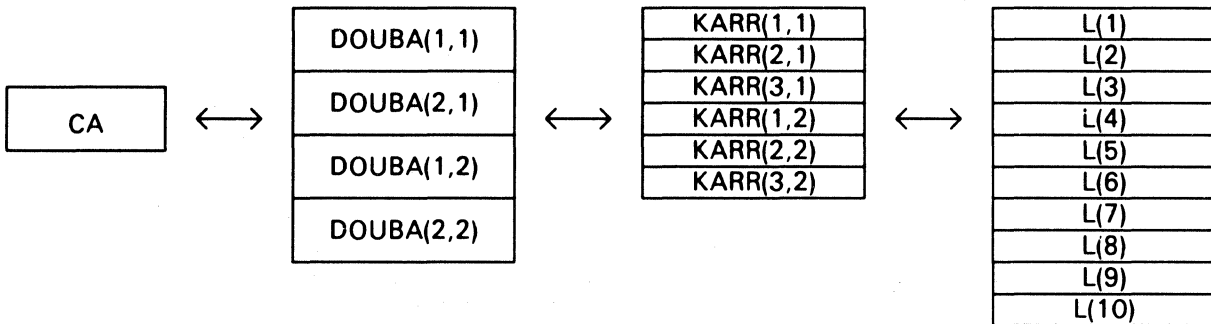
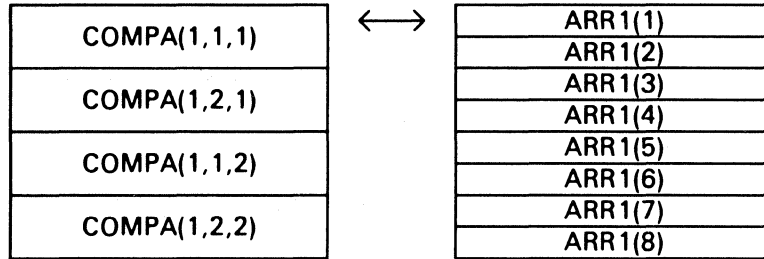
```
DOUBLE PRECISION DOUBA(2,2)
```

```
LOGICAL L(10)
```

```
EQUIVALENCE (COMPA(1), ARR1(1)),
```

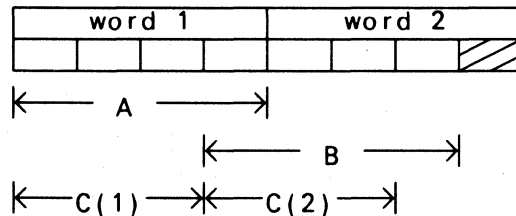
EQUIVALENCE (COMPA(1), ARR1(1))

C Based on storage word requirements, the EQUIVALENCE
C statement causes the following locations to be shared:



CHARACTER A*4, B*4, C(2)*3
EQUIVALENCE (A,C(1)), (B,C(2))

C Character variables whose names appear in equivalence have
C have the same first character storage unit. The EQUIVALENCE
C statement in this example causes the following locations
C to be shared:



C Note the fourth character position of A, the first
C character of B, and the first character position
C of C(2) all occupy the same storage location.
C The cross-hatched area is unallocated.

INTEGER I, J
CHARACTER*3 CX
EQUIVALENCE (I, CX(3:3)), (J, CX)

* Once the equivalence of I is made to the third storage
* byte of CX, the equivalence of J to the first storage
* position of CX would require J to begin on a nonword
* boundary which is in error.

6.5. COMMON Statement

Purpose:

The COMMON statement provides a means of sharing storage among various program units.

Form:

```
COMMON [ / [ c ] / ] n [ , n ] ... [ [ , ] / [ c ] / n [ , n ] ... ] ...
```

where:

c is a common block name. The variable *c* must satisfy the definition of a symbolic name (see 2.2.2). If the first *c* is omitted, the first two slashes are optional.

n is a variable name, an array name, or an array declaration.

Description:

The COMMON statement allows the user to name a storage area (a common block) to be shared by program units and to specify the names of variables and arrays which are to occupy this area. This permits:

- Different program units to refer to the same data without using arguments.
- Unrelated data from different program units to share the same storage.
- A single allocation of storage for variables and arrays, used by several program units.

The variables and arrays which appear in the list following a common block specification are declared to be in that common block.

If a common name, *c*, is omitted, blank common is assumed. Otherwise, the common block is a labeled common identified by the name *c*. There may be only one blank common block in an executable program. However, there may be many labeled common blocks.

A labeled or blank common specification may appear more than once in a COMMON statement or may appear in more than one COMMON statement. The list of variables and arrays following such successive appearances of a common block name in a program unit is treated as a continuation of the list for that block name.

Within a program unit, variables and arrays in a common block are allocated consecutive storage locations in the same order as they appear in the list. Therefore, the order in which they are entered is significant when items in the common block are referenced in other program units.

Equivalencing of two quantities that appear in COMMON statements is not possible since, by their appearance in a COMMON statement, storage is allocated for each.

Character type entities may appear in the same common block as noncharacter type entities.

Dimensions of an array to be placed in a common block may be declared in a type statement, in a DIMENSION statement, or simply by appending the dimension information to the array name in the COMMON statement itself. However, the array must not be dimensioned more than once.

For a description of storage allocation on common variables, see 6.9.1.

A COMMON statement in an internal subprogram redefines (that is, is not a continuation of) the same common block declared in the external program unit. For a description of local-global rules for names used in COMMON statements in internal subprograms, see 7.11.

FORTRAN 77 lists the following differences between labeled common and blank common:

- Execution of a RETURN or END statement sometimes causes entities in labeled common blocks to become undefined, but never causes entities in blank common to become undefined. For ASCII FORTRAN, entities in labeled common blocks do not become undefined with the execution of a RETURN or END statement.
- Labeled common blocks of the same name must be of the same size in all program units of an executable program in which they appear, but blank common blocks may be of different sizes. **ASCII FORTRAN does not put this size restriction on labeled common blocks.**
- Entities in labeled common blocks may be initially defined by means of a DATA statement in a BLOCK DATA subprogram, but entities in blank common must not be initially defined. ASCII FORTRAN allows entities in both labeled common and blank common blocks to be initialized in any program unit.

Examples:

```
COMMON A,B,C(5,5) /COM1/D,M,V,S
C      Variables A, B and array C are placed in the blank
C      common block. Variables D, M, V, and S are placed in
C      the common block named COM1.
```

```
COMMON L/C1/X,Y,Z //M,N
C      Variables L, M, and N are placed in the blank common
C      block. Variables X, Y, and Z are placed in the
C      common block named C1.
```

6.6. BANK Statement

Purpose:

The BANK statement is used to construct a multibank program. It informs the compiler of the name of the bank containing data, functions or subroutines referenced in the program unit.

Form:

```
BANK /b/ [&]n [, [&]n] ... [/b/ [&]n [, [&]n] ...] ...
```

where:

each *b* is a bank name. It must satisfy the definition of a symbolic name, and is the name specified on a Collector IBANK or DBANK directive.

each *n* is the name of a subprogram or the name of a labeled common block which is included in the associated bank, *b*. A subprogram name must be preceded by an ampersand (&). A labeled common block name must not be preceded by an ampersand.

Description:

The compiler uses this information to generate linkage between multibank subprograms and data. Note that the compiler does not actually construct the banks, it merely generates linkage between banks.

The COMPILER (BANKED=ALL) statement is much easier to use and more flexible than the BANK statement (see 8.5). However, the use of the BANK statement will result in more efficient code for programs using multiple D-banks and can be used to tune a FORTRAN program once it is debugged.

A labeled common block name and a subprogram name may not occur in the same bank. All labeled common blocks are assumed by the compiler to be in data banks and, therefore, are activated by LDJ or LBJ instructions. Subprograms are assumed to be in instruction banks and are accessed by LIJ instructions.

A subprogram name need not appear in a BANK statement if the COMPILER (LINK = IBJ\$) statement is used (see 8.5). (The COMPILER (LINK=IBJ\$) statement supersedes the use of the BANK statement on subprogram names.) However, they may be used together if desired, and the BANK name supplied will be used to link to the subprogram.

A bank name may appear more than once in a BANK statement or in more than one BANK statement. In either case, the effect is as though all the associated subprogram names or common block names were combined into one list.

A FORTRAN V subprogram name cannot appear in a BANK statement (see 7.2.3).

An internal subprogram name should not appear in a BANK statement.

A dummy subprogram name cannot appear in a BANK statement. Use the COMPILER statement option BANKED=DUMARG, LINK=IBJ\$, or BANKED=ALL to inform the compiler of a banked dummy subprogram.

A subprogram name used by an external program unit should not appear in a BANK statement in one of its internal program units.

An intrinsic function name should not appear in a BANK statement unless it was in an EXTERNAL statement first.

If a common block is banked, it should be in BANK statements in all of the program units the COMMON statements are in, or they should all have the BANKED=ALL COMPILER statement option.

In addition to the information present in the BANK statement, the compiler requires information concerning:

- The type of return from a subprogram (if a SUBROUTINE or FUNCTION is being compiled),
- The presence of bank information in dummy arguments to subprograms, and
- The presence of bank information in calling sequences of subprograms.

This information is specified by means of a COMPILER statement (see 8.5).

A comprehensive description of multibanking features can be found in the EXEC Programmer Reference, UP-4144.2 (see Preface), and the Multibanking Programmer Reference, PUP-8722.P1 (see Preface).

Examples:

```
BANK /BK/A,B,C
C      This statement indicates that common blocks A, B, and
C      C are found in the data bank named BK.
BANK /BANKA/&SUB1,&FUNC1/MRL/CBA,X,ADR
C      This statement indicates that subprograms SUB1 and FUNC1 are
C      found in the instruction bank named BANKA. Common blocks
C      CBA, X and ADR are found in the data bank named MRL.
```

6.7. PARAMETER Statement

Purpose:

The PARAMETER statement allows constants to be referred to by symbolic names. This facilitates the updating of programs in which the only changes between compilations are in the values of certain constants. The PARAMETER statement can be revised instead of changing the constants throughout the program.

Form:

```
PARAMETER ( n = e [ , n = e ] ... )
```

or

```
PARAMETER n = e [ , n = e ] ...
```

where:

n is any symbolic name (identifier). It is called a parameter constant or the symbolic name of a constant.

e is a constant expression.

Description:

The PARAMETER statement is nonexecutable. FORTRAN 77 requires a PARAMETER statement to appear before any DATA statements, statement function statements, and executable statements. ASCII FORTRAN allows the PARAMETER statement to appear anywhere in a program. However, a PARAMETER statement defining a parameter constant must appear physically before that parameter constant is referenced.

If the symbolic name *n* is of type integer, real, double precision, complex, or double precision complex, the corresponding expression *e* must be an arithmetic constant expression. If the symbolic name *n* is of type character or logical, the corresponding expression *e* must be a character constant expression or a logical constant expression, respectively.

If a symbolic name of a constant is not of default implied type, its type must be specified by a type statement or IMPLICIT statement prior to its first appearance in a PARAMETER statement. If the length specified for *n* is not the standard length as specified in Table 6-1, its length must be specified in a type statement or IMPLICIT statement prior to the first appearance of the symbolic name of the constant. Its length must not be changed by subsequent statements, including IMPLICIT statements.

The expression *e* can be any expression which yields a constant result at compile time. The operands used in *e* can be any literal constants, parameter constants which have been previously defined by other PARAMETER statements, or any FORTRAN-supplied inline or library functions. A parameter constant must not be redefined within the same program unit. The definition of parameter constants occurs from left to right within a PARAMETER statement.

FORTRAN 77 restricts *e* to a constant expression. Intrinsic function calls are not allowed. Exponentiation is allowed only if the exponent is type integer.

Note that the compiler may use the common-banked Common Mathematical Library (CML) to do compile-time arithmetic. Therefore, a PARAMETER statement requiring a CML call is not allowed at a site if the compiler is generated with the clause "USING NO MATH BANKS" on the "COMPILER" System Generation Statement (SGS) supplied to the ASCII FORTRAN compiler build.

PARAMETER statements requiring compile-time calls to the common mathematical routines are:

```
PARAMETER (IB=2**18)
PARAMETER (A=SIN(1.8))
```

Upon reference to a parameter constant in a FORTRAN source program, the constant value of that parameter constant is used by the compiler. No storage is assigned to parameter constants whose values can be represented in 18 bits or less.

For a description of local-global rules for names used in PARAMETER statements in internal subprograms, see 7.11.

Examples:

```
INTEGER A,B
PARAMETER (A=3)
PARAMETER (B=4*A)
C=A+I*B
```

C A and B have values 3 and 12 respectively at compile
C time. The third statement is interpreted as though
C the statement C=3+I*12 was written there.

```
PARAMETER D=(SIN(5.))+4)
```

C SIN(5.) will be evaluated at compile time, and D is assigned
C this value plus 4, yielding 3.0410757.

A parameter constant cannot be used as a constant in a FORMAT statement, or as a length specification in a typing statement (except for the CHARACTER type statement, where a constant expression is allowed in parentheses).

6.8. Initial Value Assignment

Initial values may be assigned to variables, arrays, or array elements in the main program or any subprogram. Initial values are loaded with the executable program. Only variables to which initial values are assigned are defined at the start of execution; all other variables are undefined until values are assigned to them. Initial values may not be specified for the formal arguments of a subprogram.

Initial value assignment is accomplished through the use of DATA statements and of initial value lists in DIMENSION and type statements. For variables and arrays in common blocks, initial values may be assigned in a BLOCK DATA subprogram (see 7.8).

A data item whose relative address under its location counter in the relocatable element is over 65535 cannot be initialized via a DATA, DIMENSION, or typing statement initialization. All local variables go under location counter zero or eight, and each common block gets its own location counter. Attempting to initialize an entity whose relative address is over 65535 (0177777 in octal) results in an ERROR 010 (see Appendix D).

6.8.1. DATA Statement

Purpose:

The DATA statement initializes variables, arrays, array elements, and character substrings at compile time.

Form:

```
DATA variable-list /const-list / [[,variable-list /const-list / ] ...
```

Description:

The *variable-list* is a list of variables, arrays, array elements, substring names, and implied-DO groups, separated by commas. The format of an implied-DO group is:

```
(d-list , index = start , stop [ ,inc ] )
```

The *d-list* is a list of array element names and implied-DO groups.

The *index* is a scalar integer variable, called the implied-DO-variable. *Start*, *stop*, and *inc* are integer constant expressions. If implied-DO groups are nested, the *start*, *stop*, and *inc* expressions of an inner group may include references to the *index* of an outer group.

An iteration count and the values of the implied-DO-variable are established from *start*, *stop*, and *inc* exactly as for a DO-loop, except that the iteration count must be positive (see 4.5.4.2). The list items in *d-list* are specified once for each iteration of the implied-DO list with the appropriate substitution of values for any occurrence of the implied-DO-variable index.

Subscript expressions in a *d-list* must be integer expressions involving only constants, parameter constants, and implied-DO index variables. However, subscript expressions within a *variable-list* but not within a *d-list* are limited to integer expressions involving only constants and parameter constants. Substring expressions must be integer constant expressions.

The *const-list* is a list of constants separated by commas. Each constant in the list may optionally be preceded by "*n**", where *n* is an unsigned positive integer constant or a positive integer-valued parameter constant. This notation denotes repetition of the immediately following constant *n* times.

The elements in the *variable-list* are matched to the elements in the *const-list* according to position. An array name matches *n* consecutive *const-list* items, where *n* is the number of elements in the array. The elements of the array are initialized in column-major order. FORTRAN 77 requires a one-to-one correspondence between the number of elements in *variable-list* and *const-list*. ASCII FORTRAN allows the lists to be of unequal length. If *const-list* is short, the last elements of the variable list will not be initialized. If *const-list* is too long, the last constants are ignored.

The constants in *const-list* may be of any type, including three constant types which are supported for initial values only: statement labels, octal, and Fieldata.

6.8.2. Statement Labels

A statement label in an initialization statement is written as a currency symbol or ampersand followed by one to five decimal digits. The value of the digit string, interpreted as an integer, must be the same as that of one of the statement labels in the program. The corresponding integer scalar variable is initialized to the same value that it would receive if it were assigned that statement label in an ASSIGN statement.

6.8.3. Octal Constants

An octal constant, allowed only in an initialization statement, is written as the letter O followed by a string of octal digits. If the string consists of more than twelve digits, only the last twelve digits are used. Note that it is possible to have a parameter constant whose name is the same as an octal constant. Such a name is interpreted as a parameter constant when used as an initial value. An octal constant can be used to initialize a character variable or a one-word noncharacter variable (that is, logical, integer, or single precision real). When used with a character variable, an octal constant is extended with zeros on the left to a multiple of 3 digits (9, 18, 27, or 36 bits), then padded with blanks or truncated on the right to match the length of the character variable. When used with a logical, integer, or real variable, an octal constant of less than twelve digits is extended to twelve digits with zeros on the left.

6.8.4. Fieldata Constants

A Fieldata constant, allowed only in an initialization statement, is written as a quoted or Hollerith literal immediately followed by the letter F. The characters of the literal are converted to the 6-bit Fieldata code and stored six to a word. Fieldata constants are padded with Fieldata blanks on the right, or are truncated on the right, to achieve the required length. Fieldata constants may not be used to initialize character variables.

6.8.5. Other Constants

Table 6-3 indicates the requirements for type matching between a *variable-list* item and the matching *constant-list* item. In general, except for character, statement label, octal, and Fieldata constants, the types must match. If the types do not match but can be converted, a warning is issued and the constant is converted to the type of the variable. If no conversion is possible, the initialization is not done.

Table 6-3. Data Initializations

Constant Type	Variable Type	Size	Error or Warning	Initialization Done	Conversion Done
INTEGER	INTEGER	same size		yes	no
	REAL		warning	yes	yes
	COMPLEX		warning	yes	yes
	LOGICAL		error	no	
	CHARACTER		error	no	
REAL	INTEGER	same size	warning	yes	yes
	REAL		yes	no	
	REAL	diff size	warning	yes	yes
	COMPLEX	same size	warning	yes	yes
	LOGICAL		error	no	
CHARACTER	error	no			
COMPLEX	INTEGER	same size	warning	yes	yes
	REAL		warning	yes	yes
	COMPLEX	diff size		yes	no
	COMPLEX	diff size	warning	yes	yes
	LOGICAL	same size	error	no	
CHARACTER	error		no		
LOGICAL	LOGICAL	same size		yes	no
LOGICAL	nonlogical		error	no	
CHARACTER	any type*	same size		yes	truncation or blank fill
statement number	INTEGER			yes	no
statement number	noninteger	same size	error	no	
Fielddata	CHARACTER		error	no	
Fielddata	noncharacter	same size		yes	truncation or blank fill
octal	INTEGER		same size		yes
	REAL *4			yes	no
	REAL *8	error		no	
	COMPLEX	error		no	
	LOGICAL			yes	no
	CHARACTER			yes	truncation or blank fill

* ASCII FORTRAN permits any variable type to be initialized with a character constant. FORTRAN 77 permits only character variables to be initialized by character constants.

DATA statements are nonexecutable. FORTRAN 77 requires DATA statements to follow all specification statements.

ASCII FORTRAN, on the other hand, allows DATA statements to appear anywhere in the program unit after any SUBROUTINE, FUNCTION, or IMPLICIT statement and before the last statement in the program unit. The one restriction ASCII FORTRAN places on DATA statements is that they must follow all specification statements that declare variables referred to in the DATA statement. If an array has its bounds declared in one statement and its type in a following statement, there must not be a DATA statement between the two specification statements. Because of these problems, the ASCII FORTRAN

compiler will issue a warning message whenever a DATA statement is followed by a specification statement.

FORTRAN 77 does not permit blank common entities to be initialized and further restricts the initialization of entities in labeled common to only appear within a BLOCK DATA subprogram. ASCII FORTRAN, on the other hand, permits data initialization of entities in labeled and blank common to appear in any program unit.

A variable, array element, or substring must not be initially defined more than once in an executable program. Note that storage association from the use of EQUIVALENCE could also result in incorrect multiple initialization. Care must be taken when initializing character type entities in the same common block from more than one program unit. If the program units are compiled as separate relocatable elements and both initialize different character type items, the initialization of some items may be destroyed during the collection process if the initialized items share the same word of storage. Character items in common are packed in storage and it is possible to have more than one item occupying the same word of storage. For this reason, initialization of character type items in common blocks should be limited to one program unit in the FORTRAN program.

Within a program unit care must be taken when initializing character type entities. Multiple initializations of the same item or parts of the same item may result in some of the initialization information being lost. Substring initializations only initialize the character positions specified. Other character positions within the same word and part of the same variable or array element will have a value of binary zero unless initialized by other substring initializations.

Examples:

```
DIMENSION C(10)
DATA A, B/1.2,0.5/, C/10*1.0/
C      The constants 1.2 and 0.5 are assigned to A and B,
C      respectively, at compile time. Each element in array C
C      is assigned a value of 1.0.
```

```
DIMENSION IARY(10), LARY(10)
DATA (IARY(I),I=1,10) /1,2,3,7*4/, (LARY(I),I=1,9,2) /5*0.0/
C      The array IARY is initialized by an implied-DO.
C      The first three elements in the array have values
C      1, 2, and 3 respectively. The remaining seven elements in
C      that array have 4 as their value. The elements LARY(1),
C      LARY(3), LARY(5), LARY(7), and LARY(9) are initialized
C      to 0.0. The other elements of LARY are not initialized.
```

```
PARAMETER (O2 = 5)
DATA I/O2/
C      O2 is a parameter constant which has 5 as its value.
C      O2 in the DATA statement is interpreted as the
C      parameter constant and the integer constant 5 is assigned
C      to the variable I, rather than the octal constant O2 (see
C      2.2.1.6).
```

```
DIMENSION A(10)
```

```
DATA A/'AB', 'CDE', 'FGHIJ' /
```

C The first three elements in array A have values
C 'ABΔΔ', 'CDEΔΔ', and 'FGHI', respectively.

```
REAL C2(10)/10*98.6/
```

C This example shows initialization in a type statement. All
C elements of array C2 are assigned the value 98.6.

```
DATA J /'ABCDEF'F/, K /0060710111213/
```

C The integer variables J and K receive
C the same initial value, J as a Fielddata constant
C and K as an octal constant.

```
CHARACTER*1 S1
```

```
DATA S1(1:1) /'*' /
```

```
CHARACTER*2 S2
```

```
DATA S2(1:1), S2(2:2) /' ', '*' /
```

```
CHARACTER*3 S3
```

```
DATA S3(1:2), S3(3:3) /' ', '*' /
```

```
CHARACTER*4 S4
```

```
DATA S4(1:3), S4(4:4) /' ', '*' /
```

C This example shows character
C variables of length n being
C initialized to n-1 blanks followed
C by an asterisk.

```
CHARACTER*1 S1 /' ' /
```

```
DATA S1(1:1) /'*' /
```

```
CHARACTER*2 S2 /' ' /
```

```
DATA S2(2:2) /'*' /
```

```
CHARACTER*3 S3 /' ' /
```

```
DATA S3(3:3) /'*' /
```

```
CHARACTER*4 S4 /' ' /
```

```
DATA S4(4:4) /'*' /
```

C This example appears to show
C character variables of length n
C being initialized to n-1 blanks
C followed by an asterisk. However,
C since there are multiple initializations
C involving parts of the same variable
C (which is not allowed by the standard),
C some initialization information is
C lost. The results are unpredictable and may
C result in some character positions being
C initialized to null characters (ASCII octal
C code 000). Note, on some output devices
C nulls are dropped giving the appearance
C of characters having been placed in
C the wrong position.

6.9. Storage Assignment

The FORTRAN compiler automatically allocates storage for program data and compiler-generated instructions.

The ordinary operation of the compiler assumes that data and program addresses in the generated object program will be less than 65536. This, of course, implies that the size of data storage for any FORTRAN program should be less than 65,536 words. However, with the O option on the processor call statement, the compiler will assume that all data addresses in the program may extend to 262,143 words. This permits the creation of much larger FORTRAN programs (see 10.5).

This facility may be used without restriction regarding the type (scalar or array, common or noncommon) of data involved. Several points should be considered when using this facility:

- It is only necessary if the last address of the program is greater than 65535 (0177777 octal) words after collection. Note that if automatic storage is used and an ER MCORES is done by storage management creating addresses greater than 65,535, the O option is not needed.
- If the total collected program size will exceed 65,535 words, all FORTRAN program units present within the program should be compiled with the O option.
- This facility should be used only if required since more instructions may be necessary to access arguments and arrays when the O option is used.
- If truncation messages naming the FORTRAN run-time routines are emitted during collection of the user absolute, IN statements may be necessary in the Collector symbolic, naming the user's own elements and common block names. This should result in the run-time routines being collected at addresses under 65,536.

If large programs (including those using a total of more than 262,143 words) are required, they may be structured into smaller components using the BANK statement (see 6.6), and a multibanked Collector symbolic (see Appendix H).

6.9.1. Data Storage Assignment

The ASCII FORTRAN system assigns storage to user noncharacter type variables in word increments. Table 6-4 details the amount of storage allocated by variable type in bytes and the alignment of the variable within that storage.

The first item in a common block list is allocated storage on a word boundary. Subsequent character items in common are allocated storage in a packed form on byte boundaries which may or may not begin on word boundaries. Packed form means there are no unallocated character storage positions or bytes between two entities in common. The common block structure determines the alignment of its members. **Equivalencing character items in common to noncharacter items is only permitted when the item in common is word-aligned. If a noncharacter item follows a character item in common, there may be one or more unallocated bytes between the two items since noncharacter data items must begin on a word boundary.**

Table 6-4. Storage Alignment and Requirement

Type	Length	Storage	Alignment
INTEGER	4	1 word	word boundary
REAL	single (4)	1 word	word boundary
	double (8)	2 words	word boundary
COMPLEX	single (8)	2 words	real part in first word
			imaginary part in second word
	double (16)	4 words	real part in first 2 words
			imaginary part in second 2 words
LOGICAL	4	1 word	word boundary (only least significant bit is used)

Character items not equivalenced and not in common are allocated storage on word boundaries except for the following cases:

- Character items of length 2 are allocated storage on word and half-word boundaries.
- Character items of length 1 are allocated storage on byte boundaries.

Character arrays, in common or not, are allocated storage such that there are no unallocated bytes between the elements of the array.

Storage order and alignment for items not in common is not determined by their appearance in the program. Storage is allocated by traversing a chain of variables whose order is determined by a hash function. The FORTRAN programmer should not try to predict storage order or alignment.

Note that the COMPILER statement option `STD=66` causes all character items, including array elements, to begin on word boundaries.

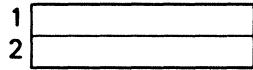
Examples:

REAL E(5)

1	E(1)
2	E(2)
3	E(3)
4	E(4)
5	E(5)

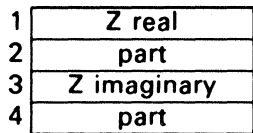
Array E occupies five words.

DOUBLE PRECISION D



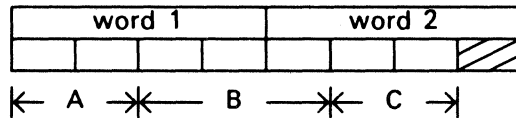
Variable D occupies two words.

COMPLEX*16 Z



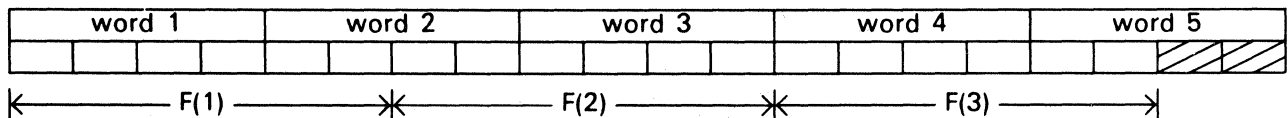
Variable Z occupies four words.

CHARACTER*2 A, B*3, C
COMMON /C1/ A, B, C



Since A, B, and C are scalars in common, they are allocated storage in a packed form. The cross-hatched area is unallocated.

CHARACTER*6 F(3)



Assuming array F begins on a word boundary, F is allocated five words of storage in a packed form. The cross-hatched area could be allocated to character items of length 1 or 2 or the area could be allocated to items in common or equivalenced to F.

6.9.2. Location Counter Usage

The instructions and data generated by the ASCII FORTRAN compiler are allocated by function to several location counters. The location counters used and their function are described in Table 6-5. For a description of the automatic storage feature, see 8.5.1 (COMPILER statement options DATA=AUTO and DATA=REUSE).

Table 6-5. Location Counter Usage

Compiler Options			
Location Counter	No Automatic Storage or O Option	O Option Without Automatic Storage	Automatic Storage
0	local (noncommon) variables	local variables	not used
1	all program instructions not resulting from I/O lists	instructions	instructions
2	blank common	blank common	blank common
3	INFO-010 diagnostic tables (see 10.7.2)	diagnostic tables	diagnostic tables
4	I/O packets, I/O list instructions, FORMAT statement text, NAMELIST statement text	I/O packets and list instructions, FORMAT and NAMELIST text, parameter lists, constants	not used
5	not used	not used	automatic-storage initialization literals
6	parameter lists	not used	not used
7	not used	not used	automatic-storage initialization code
8	not used	not used	local variables appearing in SAVE statement
9	not used	not used	not used
10	constants	not used	not used
11 and greater	labeled common	labeled common	labeled common

7. Function and Subroutine Procedures

7.1. Procedures

The term procedure refers to a code mechanism which performs a particular computation. FORTRAN supports three general types of procedures:

- A function procedure which is invoked by the appearance of the function name followed by its parameter list in an expression. A function returns a value which replaces the name in the evaluation of the expression.
- A subroutine procedure which is invoked by the appearance of its name in a CALL statement. A subroutine does not return a value, although it may modify the values of variables.
- A main program procedure which is invoked by action of the operating system, usually as a result of the @XQT Executive control statement.

Subroutines and functions are referred to collectively as subprograms.

A program consists of a main program and zero or more subprograms.

Subprograms may be external or internal. One or more program unit groups may be compiled under one processor call card (@FTN). A program unit group is composed of the external program unit and any internal subprograms contained in the external program unit. For example:

```
@FTN, IS      NAME
      PROGRAM MAIN
      .
      .
      .
      END
      SUBROUTINE IN
      .
      .
      .
      END
      FUNCTION SOLVE
      .
      .
      .
      END
@EOF
```

An internal subprogram is considered as nested within the previous external subprogram, and may access the external subprogram's data as well as having its own local data. An external subprogram may have many internal subprograms. See 7.4.2 and 7.4.3 for a more detailed description of internal and external subprograms.

The source input to the ASCII FORTRAN compiler may consist of one or more program units (see 10.2.1). If a main program is present in the source input, then it must physically be the first program unit in the input.

The relocatable binary elements produced by the ASCII FORTRAN compiler (or any other SPERRY UNIVAC Series 1100 Operating System language processor) are combined into an executable absolute element by the Collector.

In addition to subroutines and functions, FORTRAN supports a third type of subprogram, the BLOCK DATA subprogram. Such a subprogram contains no executable code and is used solely for the assignment of initial values to variables in COMMON blocks. A BLOCK DATA subprogram cannot be called. BLOCK DATA subprograms are described in 7.8.

7.2. Procedure References

The code represented by a procedure name is executed when the name of the procedure is encountered. This name is followed by an actual parameter list. If a function has no parameters, it is referred to with a void parameter list, that is, (). Subroutine references must follow the keyword CALL in a CALL statement. If the subprogram is used as an argument for another subprogram, it must appear previously with an explicit actual argument list or be identified in an EXTERNAL statement (see 7.2.3). When a subprogram is used as an argument, it is never followed by a parameter list and the code associated with the procedure is not executed at this point. If an intrinsic function is used as an argument for another subprogram, it must be identified in an INTRINSIC statement (see 7.2.4).

The following paragraphs describe function references (except statement functions, which are described in 7.4.1.2), subroutine references, and the EXTERNAL and INTRINSIC statements.

7.2.1. Function References

A function is referenced by the appearance of its name with an explicit actual parameter list. The form of a function reference is:

$$f ([a [, a] \dots])$$

where f is the name of the function. (The name f must not appear both as the name of a function and as the name of a subroutine in the same program unit.) Each a is an actual argument and must match the corresponding formal parameter of f in type and usage (see 7.5). The arguments can be FORTRAN expressions, array names, function (intrinsic function, external procedure, or dummy procedure) names, or statement labels. The number of arguments may not exceed 150.

When a statement label appears as an actual argument, it is written as $*n$, $&n$, or $\$n$, where n is the statement label. The corresponding formal parameter must be an asterisk (*) or currency symbol (\$). (See 7.6.) Any alternate returns will result in a loss of the value normally returned by a function reference.

A reference to a function causes the computation indicated by the function definition to be performed. The value returned by the function is used to determine the value of the expression in which the reference occurs. The value returned is assumed to be of the type indicated by the first character of the name f , unless f has appeared in a type statement, or unless the first character of f has appeared in an IMPLICIT statement.

If an external procedure name appears in an actual argument list without an explicit actual argument list, the procedure name is passed to the called function. If an external procedure name appears without an explicit actual parameter list, it must have appeared previously with an explicit actual argument list or in an EXTERNAL statement (see 7.2.3).

7.2.2. Subroutine References

A subroutine is referred to by the appearance of its name following the keyword CALL in a CALL statement. The form of a CALL statement is:

```
CALL s [ ( [ a[ , a] ... ] ) ]
```

where *s* is the name of the subroutine. (The name *s* may not appear both as the name of a subroutine and as the name of a function in the same program unit.) Each *a* is an actual argument and must match the corresponding formal parameter of *s* in type and usage (see 7.5). The arguments can be FORTRAN expressions, array names, function (intrinsic function, external procedure, or dummy procedures) names, or statement labels. The number of arguments may not exceed 150.

When a statement label appears as an actual argument, it is written as **n*, *&n*, or *\$n*, where *n* is the statement label. The corresponding formal parameter must be an asterisk (*) or currency symbol (\$). (See 7.6.) External procedure names are used in the same manner as function references (see 7.2.1).

A reference to a subroutine results in execution of the body of the subroutine. When execution of the subroutine is complete, the calling routine is resumed at the statement following the call, or at one of the statement labels appearing in the parameter list of the CALL statement.

7.2.3. EXTERNAL Statement

Purpose:

The EXTERNAL statement informs the compiler that a programmer-written subprogram exists external to the program unit.

Form:

```
EXTERNAL a[ , a] ...
```

or:

```
EXTERNAL [opt] a [ ( / ) ] [ , [opt] a [ ( / ) ] ] ...
```

where:

a is a symbolic name.

opt is & or *; & has no meaning and is provided for syntactic compatibility with other FORTRAN processors. The * indicates that *a* is a FORTRAN V external subprogram, and is also retained for compatibility.

/ is FOR, ACOB, or PL1.

Description:

The EXTERNAL statement establishes that each *a* is a subprogram (that is, function or subroutine) name, and is not a FORTRAN-supplied function or a variable. This statement is required if the first reference to *a* (excluding the appearance of *a* in other specification statements) is without an explicit actual argument list. Thus a subroutine or function name which is not called, but only passed as an actual argument, must appear in an EXTERNAL statement. This holds for internal subprogram names also.

The FORTRAN 77 standard specifies that any user-supplied function or subroutine name passed as an argument must appear in an EXTERNAL statement.

If a name *a* in an EXTERNAL statement is the same as a FORTRAN-supplied intrinsic function name, the name *a* will be treated as a user-supplied subprogram name. The name *a* loses all properties which are associated with the FORTRAN-supplied function: type, automatic typing, required type and number of parameters. If *a* is referenced as a function, its type will be determined from the first letter of the name, unless *a* appears in an explicit type statement, or unless the first character of *a* appears in an IMPLICIT statement.

The optional keyword / following each name *a* may be FOR, ACOB, or PL1. It indicates that *a* is a FORTRAN V, ASCII COBOL, or PL/1 external subprogram, and enables ASCII FORTRAN to generate the correct linkage for each language.

The EXTERNAL statement is a specification statement and must precede any executable statements. In addition, any reference to a name *a* in a statement function definition, other than as a formal parameter of the statement function, must follow the EXTERNAL statement.

For a description of local-global rules for names used in EXTERNAL statements in internal subprograms, see 7.11.

Example:

```
EXTERNAL COMP1, COMP2
.
.
CALL PROG (COMP1, VAR1)
.
.
CALL PROG (COMP2, VAR2)
.
.
END
```

C The first call passes the name of subprogram COMP1
C as an argument of subroutine PROG. The second call
C passes the name of subprogram COMP2.

7.2.4. INTRINSIC Statement

Purpose:

An INTRINSIC statement is used to identify a symbolic name as representing an intrinsic function (see 7.3.1). It also permits a name that represents a specific intrinsic function to be used as an actual argument.

Form:

```
INTRINSIC f [ , f ] ...
```

where *f* is an intrinsic function name.

Description:

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 names of intrinsic functions for type conversion (INT, IFIX, HFIX, IDINT, IDFIX, FLOAT, REAL, DBLE, DFLOAT, DREAL, CMPLX, DCMPLX, ICHAR, CHAR, UPPERC, LOWERC, and for choosing the largest or smallest value (MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1) must not be used as actual arguments.

Only one appearance of a symbolic name in all of the INTRINSIC statements of a program unit is permitted. Note that a symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in a program unit.

Example:

```
INTRINSIC SIN  
CALL SUB(SIN,COS)  
C The intrinsic function name, SIN, is passed as the first argument  
  to subroutine  
C SUB. The real scalar variable COS is passed as the second  
  argument.
```

7.3. FORTRAN-Supplied Procedures

The ASCII FORTRAN system provides a set of precoded procedures, called intrinsic procedures, for the convenience of the programmer. These procedures are broken down into three groups:

- Intrinsic functions
- Pseudo-functions
- Service subroutines

These groups are described in detail in the subsections which follow.

References to certain intrinsic procedures cause code to be generated which performs the requested action at the point of the reference. Such procedures are called inline procedures. References to the other intrinsic procedures result in calls to library subprograms; these procedures are called library procedures. The inline or library nature of each intrinsic procedure is specified in its respective description.

7.3.1. Intrinsic Functions

Numerous intrinsic functions (sometimes called built-in functions) are provided with the processor. These are not written or modified by the programmer. However, their actual form and manner of referencing corresponds to that of a subprogram defined by a FUNCTION statement in a source module. These functions always return one value (function value) to the calling statement and perform computations frequently needed in FORTRAN programs.

The nonmathematical intrinsic functions are usually called typeless functions. Each argument of a typeless function is a single-word expression. If the argument is a character expression whose length is less than four characters (that is, less than one word), the argument is left-justified and blank-filled before performing the function. The function is performed bit-by-bit, treating a zero in the sense of false and a one in the sense of true. These functions are summarized in Table 7-1. See 2.2.3.4.2 for more information on expressions involving typeless functions.

Table 7-1. Typeless Intrinsic Functions

Function Description	Function Name	Number of Args	Result	Sample Reference
Logical Product	AND	2	Typeless	AND(<i>word</i> , <i>word2</i>)
Logical Sum	OR	2	Typeless	OR(<i>word</i> , <i>word2</i>)
Exclusive OR	XOR	2	Typeless	XOR(<i>word</i> , <i>word2</i>)
Treat Argument as Typeless	BOOL	1	Typeless	BOOL(<i>word</i>)
Complement	COMPL	1	Typeless	COMPL(<i>word</i>)

The mathematical and character intrinsic functions are described in Table 7-2. The column entitled Proc Type in Table 7-2 indicates whether the associated reference causes code to be generated which performs the necessary computation (inline) or results in a call to a library element. In the function descriptions, the notation *pv expression* denotes the principal value of the multiple-valued complex *expression*.

An intrinsic function is referred to in an expression as if it were a single value or user function.

All trigonometric angles are expressed in radians.

The type of an intrinsic function may be declared in the program unit that contains a reference to it. However, the declared type must match the type specified for the function in Table 7-2. The intrinsic function name retains its automatic typing attribute.

If an intrinsic function name appears in an EXTERNAL statement, the name becomes an external procedure name. If an intrinsic function name appears in an explicit typing statement which differs from the type associated with the name, the name loses its intrinsic properties. The name could be a user-supplied function or a variable name, depending on the use of the name. Other names for the same function retain their automatic typing attribute.

For example, in the code:

```
INTEGER SIN, COS
Y = SIN(X) * DSIN(X) + COS
```

SIN(X) is a call to an external function which must be supplied. The reference DSIN(X) is a reference to the single precision sine function, because of automatic type association. The reference to COS is a reference to the scalar variable COS.

If a reference is made to an intrinsic function with an argument of a type or length different from that specified in Table 7-2, the ASCII FORTRAN compiler provides an automatic function selection facility. The compiler will automatically select, from the same group, the function which processes the type of argument passed. (The FORTRAN 77 standard only has this automatic function selection facility on the generic names.) Generally, if the argument is of a type for which there is no corresponding function in Table 7-2, or has a different number of arguments, the function is assumed to be user-supplied. If a reference is made to a function which does not handle an integer argument but does handle a real argument, the integer argument will be converted to type real. The MAX and MIN functions constitute a special case. If the type of the arguments does not match the type specified for the function name used, the function will be selected from MAX0/AMAX1/DMAX1 or MIN0/AMIN1/DMIN1, respectively.

If an intrinsic function name first appears in the program unit in an executable statement with no following parentheses or arguments, the name is assumed to be a scalar variable name.

When an intrinsic function is being used as an actual argument, only a specific intrinsic function name may be used. In addition, the specific intrinsic function name must have appeared in an INTRINSIC statement. See 7.2.4 for a list of intrinsic function names that cannot be used as actual arguments.

For intrinsic functions which require more than one argument, the arguments should all be of the same type. If they are not, they will be converted to their highest common type before selection of the appropriate function.

An IMPLICIT statement does not change the type of an intrinsic function.

AMOD, MOD, and DMOD are not defined when the value of the second argument is zero.

Table 7-2. Intrinsic Functions

Function Description	Function Name	Arguments		Function Value	Proc. Type	Generic Name
		No.	Type			
Natural logarithm $y = \ln x$ ($x > 0$) $y = \text{pv} \ln x$	ALOG	1	Real	Real	Library	LOG
	DLOG		Double	Double		
	CLOG		Complex	Complex		
	CDLOG		D-complex	D-complex		
Common logarithm $y = \log_{10} x$ ($x > 0$)	ALOG10	1	Real	Real	Library	LOG10
	DLOG10		Double	Double		
Exponential $y = e^x$	EXP	1	Real	Real	Library	EXP
	DEXP		Double	Double		
	CEXP		Complex	Complex		
	CDEXP		D-complex	D-complex		
Square root $y = \sqrt{x}$ ($x \geq 0$) $y = \text{pv}\sqrt{x}$	SQRT	1	Real	Real	Library	SQRT
	DSQRT		Double	Double		
	CSQRT		Complex	Complex		
	CDSQRT		D-complex	D-complex		
Arc sine $y = \text{Arcsin } x$ (y in radians) $-1 \leq x \leq 1$	ASIN	1	Real	Real	Library	ASIN
	ARSIN		Real	Real		
	DASIN		Double	Double		
	DARSIN		Double	Double		
Arc cosine $y = \text{Arcos } x$ (y in radians) $-1 \leq x \leq 1$	ACOS	1	Real	Real	Library	ACOS
	ARCOS		Real	Real		
	DACOS		Double	Double		
	DARCOS		Double	Double		
Arc tangent $y = \text{Arctan } x$ $y = \text{Arctan } (x_1 / x_2)$ (y in radians)	ATAN	1	Real	Real	Library	ATAN
	DATAN		Double	Double		
	ATAN2	2	Real	Real	Library	ATAN2
	DATAN2		Double	Double		
Sine $y = \sin x$ (x in radians)	SIN	1	Real	Real	Library	SIN
	DSIN		Double	Double		
	CSIN		Complex	Complex		
	CDSIN		D-complex	D-complex		
Cosine $y = \cos x$ (x in radians)	COS	1	Real	Real	Library	COS
	DCOS		Double	Double		
	CCOS		Complex	Complex		
	CDCOS		D-complex	D-complex		
Tangent $y = \tan x$ (x in radians)	TAN	1	Real	Real	Library	TAN
	DTAN		Double	Double		
	CTAN		Complex	Complex		
	CDTAN		D-complex	D-complex		
Cotangent $y = \text{cotan } x$ (x in radians)	COTAN	1	Real	Real	Library	COTAN
	DCOTAN		Double	Double		
Hyperbolic sine $y = (e^x - e^{-x}) / 2$	SINH	1	Real	Real	Library	SINH
	DSINH		Double	Double		
	CSINH		Complex	Complex		
	CDSINH		D-complex	D-complex		

Table 7-2. Intrinsic Functions (continued)

Function Description	Function Name	Arguments		Function Value	Proc. Type	Generic Name	
		No.	Type				
Cube root $y = x^{1/3}$ $y = px^{1/3}$	CBRT	1	Real	Real	Library	CBRT	
	DCBRT		Double	Double			
	CCBRT		Complex	Complex			
	CDCBRT		D-complex	D-complex			
Hyperbolic cosine $y = (e^x + e^{-x}) / 2$	COSH	1	Real	Real	Library	COSH	
	DCOSH		Double	Double			
	CCOSH		Complex	Complex			
	CDCOSH		D-complex	D-complex			
Hyperbolic tangent $y = (e^x - e^{-x}) / (e^x + e^{-x})$	TANH	1	Real	Real	Library	TANH	
	DTANH		Double	Double			
	CTANH		Complex	Complex			
	CDTANH		D-complex	D-complex			
Absolute value (see note 1) $y = x $ $y = ((\text{real } x)^2 + (\text{imag } x)^2)^{1/2}$	IABS	1	Integer	Integer	Inline	ABS	
	ABS		Real	Real			Inline
	DABS		Double	Double			Inline
	CABS		Complex	Real			Library
	CDABS		D-complex	Double			Library
Error function $y = (2/\sqrt{\pi}) \int_0^x e^{-u^2} du$	ERF	1	Real	Real	Library	ERF	
	DERF		Double	Double			
Complementary error function $y = (2/\sqrt{\pi}) \int_x^\infty e^{-u^2} du$ $= 1 - \text{ERF}(x)$	ERFC	1	Real	Real	Library	ERFC	
	DERFC		Double	Double			
Maximum value $y = \text{largest value from } \{x_1, \dots, x_n\}$ $y \geq x_i, 1 \leq i \leq n$	MAX0	≥ 2	Integer	Integer	Inline	MAX	
	AMAX1		Real	Real			
	DMAX1	Double	Double	Double			
	AMAX0	≥ 2	Integer	Real	Inline	—	
Minimum value $y = \text{smallest value from } \{x_1, \dots, x_n\}$ $y \leq x_i, 1 \leq i \leq n$	MIN0	≥ 2	Integer	Integer	Inline	MIN	
	AMIN1		Real	Real			
	DMIN1	Double	Double	Double			
	AMIN0	≥ 2	Integer	Real	Inline	—	
Gamma $y = \int_0^\infty u^{x-1} e^{-u} du$ For real: $0 < x < 34$ For double: $0 < x < 170$	GAMMA	1	Real	Real	Library	GAMMA	
	DGAMMA		Double	Double			
Log gamma $y = \ln \int_0^\infty u^{x-1} e^{-u} du$ ($x > 0$)	ALGAMA	1	Real	Real	Library	LGAMMA	
	DLGAMA		Double	Double			
Remaindering - remainder of dividing x_1 by x_2 $y = x_1 \text{ (modulo } x_2)$	MOD	2	Integer	Integer	Inline	MOD	
	AMOD		Real	Real			
	DMOD		Double	Double			

Table 7-2. Intrinsic Functions (continued)

Function Description	Function Name	Arguments		Function Value	Proc. Type	Generic Name
		No.	Type			
TYPE CONVERSION Conversion to integer int(a) (see note 2)	— INT IFIX HFIX IDINT IDFIX — —	1	Integer Real Real Real Double Double Complex D-complex	Integer	Inline	INT
Conversion to real (see note 3)	REAL FLOAT — SNGL — —	1	Integer Integer Real Double Complex D-complex	Real	Inline	REAL
Conversion to double (see note 4)	— DFLOAT — — — DREAL	1	Integer Integer Real Double Complex D-complex	Double	Inline	DBLE
Conversion to complex (see note 5)	— — — — —	1 or 2	Integer Real Double Complex D-complex	Complex	Inline	CMPLX
Conversion to double complex (see note 5)	— — — — —	1 or 2	Integer Real Double Complex D-complex	D-complex	Inline	DCMPLX
Conversion to integer (see note 6)	ICHAR	1	Character	Integer	Inline	—
Conversion to character (see note 6)	CHAR	1	Integer	Character	Inline	—
Truncation - int(a) (see note 2)	AINT DINT	1	Real Double	Real Double	Inline	AINT
Nearest whole number int(a+.5) if a ≥ 0 int(a-.5) if a < 0	ANINT DNINT	1	Real Double	Real Double	Library	ANINT
Nearest integer int(a+.5) if a ≥ 0 int(a-.5) if a < 0	NINT IDNINT	1	Real Double	Integer	Inline	NINT
Transfer of sign $y = x_1 $ if $x_2 \geq 0$ $= - x_1 $ if $x_2 < 0$	ISIGN SIGN DSIGN	2	Integer Real Double	Integer Real Double	Inline	SIGN
Positive difference $y = x_1 - \min(x_1, x_2)$	IDIM DIM DDIM	2	Integer Real Double	Integer Real Double	Inline	DIM

Table 7-2. Intrinsic Functions (continued)

Function Description	Function Name	Arguments		Function Value	Proc. Type	Generic Name
		No.	Type			
Imaginary part of COMPLEX argument (see note 1)	AIMAG	1	Complex	Real	Inline	IMAG
	DIMAG		D-complex	Double		
Complex conjugate (see note 1) $y = a-bi$ for $x = a+bi$	CONJG	1	Complex	Complex	Inline	CONJG
	DCONJG		D-complex	D-complex		
Double precision product $y = x_1 * x_2$	DPROD	2	Real	Double	Inline	—
Length $y =$ length of character entity (see note 11)	LEN	1	Character	Integer	Inline	—
Index of a substring $y =$ location of substring x_2 in string x_1 (see note 10)	INDEX	2	Character	Integer	Library	—
Lexically greater than or equal (see note 12)	LGE	2	Character	Logical	Inline	—
Lexically greater than (see note 12)	LGT	2	Character	Logical	Inline	—
Lexically less than or equal (see note 12)	LLE	2	Character	Logical	Inline	—
Lexically less than (see note 12)	LLT	2	Character	Logical	Inline	—
Uppercase – return a string identical to the argument, except that all lowercase ASCII characters have been converted to uppercase	UPPERC	1	Character	Character	Library	—
Lowercase – return a string identical to the argument, except that all uppercase ASCII characters have been converted to lowercase	LOWERC	1	Character	Character	Library	—
Trim length – return the length of a string which is the argument with all trailing blanks removed	TRMLEN	1	Character	Integer	Library	—

- NOTES:**
1. A complex value is expressed as an ordered pair of reals, (x_r, x_i) , where x_r is the real part and x_i is the imaginary part.
 2. For x of type integer, $\text{int}(x)=x$. For x of type real or double precision, there are two cases: if $|x| < 1$, $\text{int}(x)=0$; if $|x| \geq 1$, $\text{int}(x)$ is the integer whose magnitude is the largest integer that does not exceed the magnitude of x and whose sign is the same as the sign of x . For example:

$$\text{int}(-3.7) = -3$$

For x of type complex, $\text{int}(x)$ is the value obtained by applying the above rule to the real part of x . For x of type real, $\text{IFIX}(x)$ is the same as $\text{INT}(x)$.

3. For x of type real, $\text{REAL}(x)$ is x . For x of type integer or double precision, $\text{REAL}(x)$ is as much precision of the significant part of x as a real datum can contain. For x of type complex, $\text{REAL}(x)$ is the real part of x .

For x of type integer, $\text{FLOAT}(x)$ is the same as $\text{REAL}(x)$.

4. For x of type double precision, $\text{DBLE}(x)$ is x . For x of type integer or real, $\text{DBLE}(x)$ is as much precision of the significant part of x as a double precision datum can contain. For x of type complex, $\text{DBLE}(x)$ is as much precision of the significant part of the real part of x as a double precision datum can contain.

5. **CMPLX** may have one or two arguments.

For x of type integer, real, or double precision, $\text{CMPLX}(x)$ is the complex value whose real part is $\text{REAL}(x)$ and whose imaginary part is zero.

For x of type complex, $\text{CMPLX}(x)$ is x .

For x of type complex*16, $\text{CMPLX}(x)$ is the complex value whose real part is $\text{REAL}(x)$ and whose imaginary part is $\text{REAL}(\text{DIMAG}(x))$.

$\text{CMPLX}(x_1, x_2)$ is the complex value whose real part is $\text{REAL}(x_1)$ and whose imaginary part is $\text{REAL}(x_2)$.

DCMPLX may have one or two arguments.

For x of type integer, real, or double precision, $\text{DCMPLX}(x)$ is the complex*16 value whose real part is $\text{DBLE}(x)$ and whose imaginary part is zero.

For x of type complex*16, $\text{DCMPLX}(x)$ is x .

For x of type complex, $\text{DCMPLX}(x)$ is the complex*16 value whose real part is $\text{DBLE}(x)$ and whose imaginary part is $\text{DBLE}(\text{AIMAG}(x))$.

$\text{DCMPLX}(x_1, x_2)$ is the complex*16 value whose real part is $\text{DBLE}(x_1)$ and whose imaginary part is $\text{DBLE}(x_2)$.

6. *ICHAR provides a means of converting from a character to an integer, based on the position of the character in the ASCII collating sequence. The first character in the collating sequence corresponds to position 0 (the ASCII control character NUL, which has octal representation 000), and the last one corresponds to position 255 (octal representation 0377).*

The value of $ICHAR(x)$ is an integer in the range: $0 \leq ICHAR(x) \leq 255$, where x is an argument of type character of length one. The position of that character in the ASCII collating sequence is the value of $ICHAR$.

For any ASCII characters c_1 and c_2 , $(c_1 .LE. c_2)$ is true if and only if $(ICHAR(c_1) .LE. ICHAR(c_2))$ is true, and $(c_1 .EQ. c_2)$ is true if and only if $(ICHAR(c_1) .EQ. ICHAR(c_2))$ is true.

$CHAR(i)$ returns the character in the i^{th} position of the ASCII collating sequence. The value is of type character of length one. The parameter i must be an integer expression whose value must be in the range $0 \leq i \leq 255$.

$ICHAR(CHAR(i)) = i$ for $0 \leq i \leq 255$.

$CHAR(ICHAR(c)) = c$ for any ASCII character c .

7. *All angles are expressed in radians.*
8. *The result of a function of type complex is the principal value.*
9. *All arguments in an intrinsic function reference must be of the same type.*
10. *$INDEX(x_1, x_2)$ returns an integer value indicating the starting position within the character string x_1 , of a substring identical to string x_2 . If x_2 occurs more than once in x_1 , the starting position of the first occurrence is returned.*
- If x_2 does not occur in x_1 , the value zero is returned. Note that zero is returned if $LEN(x_1) < LEN(x_2)$.*
11. *The value (that is, the contents) of the argument of the LEN function need not be defined at the time the function reference is executed.*
12. *Using the intrinsic functions LGE , LGT , LLE , or LLT with arguments $op1$ and $op2$ (both character expressions) performs the same comparisons and returns the same result (.TRUE. or .FALSE.) as the character relational expression $op1 \text{ rel } op2$, where rel is $.GE.$, $.GT.$, $.LE.$, or $.LT.$, respectively.*

7.3.2. Pseudo-Functions

ASCII FORTRAN supplies two pseudo-functions, BITS and SUBSTR, to facilitate the manipulation of bit and character strings.

These functions are similar to intrinsic functions, except that pseudo-functions may also appear as the targets of assignment statements.

SBITS, a bit-manipulation intrinsic function, is also supplied.

BITS, SBITS, and SUBSTR are not in the FORTRAN 77 standard.

7.3.2.1. BITS and SBITS

7.3.2.1.1. BITS

Purpose:

The BITS pseudo-function selects a bit string of a specific length from an entity of any type.

Form:

BITS (*e*, *i*, *l*)

where:

- e* is an expression, unless BITS is used as the target of an assignment statement, in which case *e* must be a scalar variable or array element.
- i* is an integer expression specifying the initial bit position of the bit string to be selected, where $1 \leq i \leq t$ (*t* is the total number of bits in item *e*).
- l* is an integer expression specifying the length (in bits) of the bit string being selected, where $1 \leq l \leq 36$. If BITS is used as the target of an assignment statement, then $1 \leq l \leq t$.

Description:

When BITS appears in an arithmetic expression (that is, not used as the target in an assignment statement), it returns a 36-bit integer result, which is set up as follows (note that bits are counted from left to right, starting at 1): the rightmost *l* bits are taken from bits *i* to *i* + *l* - 1 of *e*, and the leftmost 36 - *l* bits are zero-filled.

When BITS is used as a pseudo-function (target of an assignment), *e* must be a scalar variable or array element. The value from the right side of the assignment is stored in the indicated bits of *e*, and all other bits of *e* are unchanged. More than 36 bits of *e* may be set. Call the right side expression *r*, with length *l*₁ (in bits). If *l*₁ > *l*, then only the rightmost *l* bits of *r* are used. If *l*₁ < *l*, then all of *r* is placed in *e* starting at bit *i* (that is, left justified, in bits *i* to *i* + *l*₁ - 1) and the remaining portion of *e* to be filled (bits *i* + *l*₁ to *i* + *l* - 1) contains zeros.

The relational $i + l - 1 \leq t$ must be satisfied for all uses of BITS.

Examples:

```
DATA L / 000022000033/  
L1 = BITS(L,1,18)    @L1 is 18 (octal 022)  
  
N = 64                @octal 0100  
BITS(N,31,6) = 63    @N is 127 (octal 0177)
```

```
CHARACTER C8*8/'abcdefgh'/. C4*4/'1234'/  
BITS(C8,37,18) = 'Z'
```

* The fifth character of C8 gets 'Z', and the sixth character
* gets all zero bits.

```
BITS(C4,19,18) = 'abc'    @C4 is '12ab'
```

```
COMPLEX*16 C16  
DOUBLE PRECISION R8  
BITS(C16,73,72) = R8
```

* R8 is placed in the imaginary portion of C16 (3rd and
* 4th words)

```
BITS(C16,1,144) = R8
```

* R8 is placed in the real portion of C16 (first 2 words),
* with zero in the imaginary part (3rd and 4th words).

7.3.2.1.2. SBITS**Purpose:**

The SBITS intrinsic function is similar to BITS, except that the result is sign-extended.

Form:

```
SBITS (e, i, l)
```

where:

- e* an expression, is the entity that the bit string is selected from.
- i* an integer expression, is the initial bit position in *e* of the bit string being selected ($1 \leq i \leq t$, where *t* is the total number of bits in item *e*).
- l* an integer expression, is the length in bits of the bit string being selected ($1 \leq l \leq 36$).

Description:

SBITS is a sign-extended BITS function, with a 36-bit integer result. It may not be used as a pseudo-function.

The 36-bit result is set up as follows (note that bits are counted from left to right, starting at 1): the rightmost *l* bits are taken from bits *i* to *i+l-1* of *e*, and the leftmost 36-*l* bits are all set to bit *i* of *e* (the sign bit of the extracted bit string). (Note that the BITS function sets the leftmost 36-*l* bits of the result to zero.) The relational $i+l-1 \leq t$ must be satisfied.

The sign-extension feature of SBITS allows results to be negative numbers. For example, if several signed integers were packed into a single word, SBITS could properly extract individual signed (positive or negative) integers.

Examples:

```
DATA N /0767574737271/  
I = BITS(N,31,6)    @I will be 57 (octal 071)  
J = SBITS(N,31,6)  @J will be -6 (octal 077777777771)  
  
CHARACTER*10 /'1234567890'/  
I = BITS(A,76,6)    @I will be 57 (octal 071)  
J = SBITS(A,76,6)  @J will be -6
```

7.3.2.2. SUBSTR**Purpose:**

The SUBSTR pseudo-function selects a character substring from a given character string.

Form:

SUBSTR (*e*, *i*, *l*)

where:

- e* is a character expression, unless SUBSTR is used as the target of an assignment statement, in which case *e* must be a scalar character variable or character array element.
- i* is an integer expression specifying the initial character position of the substring being selected, where $i \geq 1$ and *i* is less than or equal to the length of *e*.
- l* is an integer expression specifying the length (in characters) of the substring being selected, where *l* has the same restrictions as *i*.

Description:

The SUBSTR pseudo-function may be used in any character or relational expression.

The function selects a character string which is *l* characters long starting at position *i* of expression *e*. Characters are counted from left to right, so if *i* is one, the substring would start with the first character of expression *e*. The expression $i + l - 1$ (representing the character position of the final character of the substring) must be less than or equal to the length of *e*.

This function may also be used as the target of an assignment statement. However, *e* must be a simple variable, an array element or a statement function reference for which assignment is permitted. The value from the right side of the assignment is stored in the indicated substring. All other characters of the target remain unchanged.

Note that substrings may also be specified using the colon syntax specified in 2.2.2.5.

Examples:

```
C      If STR1 is typed character, then  
C      STR1 = SUBSTR('NEWYEARRESOLUTION',8,10)  
C      will select substring 'RESOLUTION' to be assigned  
C      to STR1.  
  
C      If YEAR is character and contains the value '1973'  
C      SUBSTR(YEAR,4,1) = '4'  
C      will leave YEAR with a value of '1974'.
```

7.3.3. Service Subroutines

SPERRY UNIVAC Series 1100 ASCII FORTRAN provides a set of service subroutines for the programmer's convenience and information. Some of these subroutines aid the programmer in pinpointing problem areas (DUMP and PDUMP). Others check for specific error conditions or convert information between ASCII and Fielddata forms, while another stops execution at any point indicated by the programmer (EXIT). ERTRAN allows the programmer to refer to certain Executive Request functions. Another set allows the programmer to capture and diagnose arithmetic exceptions at execution time.

Service subroutines are not defined by the programmer, but are similar in form to a subroutine subprogram in a FORTRAN source module. These functions may or may not return a value to the calling statement. Each service subroutine is explained individually in the paragraphs which follow. Service subroutines are not treated in a special manner by the compiler, but rather are treated exactly the same as a user program. Therefore, if a service subroutine can be called both as a subroutine and as a function, only one type of reference can be used from a given program unit. The use of service subroutines cannot be diagnosed by the compiler as nonstandard, since they are looked upon as user-supplied subprograms. Their names are not treated as special names by the compiler.

7.3.3.1. DUMP

Purpose:

The DUMP subroutine dumps the contents of specified storage to the system output file and terminates execution.

Form:

CALL DUMP (*a*, *b*, *f*)

where:

a and *b* are variables, array elements, or arrays that indicate the limits of storage to be dumped.

f indicates the dump format.

Description:

A dump of storage contents between *a* and *b*, inclusive, is made according to format *f* and execution is then terminated.

Either *a* or *b* may be the upper or lower limit of storage; but both must be in the same program unit or the same common block.

The format, *f*, may be any of the following:

<u>Code</u>	<u>Format Indicated</u>
0	Octal
1	Not used
2	LOGICAL
3	Not used
4	INTEGER
5	REAL
6	DOUBLE PRECISION
7	COMPLEX
8	COMPLEX*16
9	CHARACTER

Example:

```
      CALL DUMP (LOWER, UPPER, 0)
C          This statement indicates that storage should be dumped
C          in octal format starting at location LOWER and ending
C          with location UPPER. Execution is then terminated.
```

7.3.3.2. PDUMP

Purpose:

The PDUMP subroutine dumps the contents of specified storage to the system output file and continues execution.

Form:

```
CALL PDUMP ( a, b, f )
```

where:

a and *b* are variables, array elements, or arrays that indicate the limits of storage to be dumped.

f indicates the dump format. The format possibilities are the same as for the DUMP subroutine (see 7.3.3.1).

Description:

A dump of storage contents between *a* and *b*, inclusive, is made according to format *f* and program execution is resumed.

Either *a* or *b* may be the upper or lower limit of storage. Both must be in the same program unit or the same common block.

Example:

```
      CALL PDUMP (LOWER, UPPER, 5)
C      A dump is made of information between locations
C      UPPER and LOWER. This data is output in single
C      precision real format to the system output file and
C      normal program execution is then resumed.
```

7.3.3.3. DVCHK**Purpose:**

The DVCHK subroutine tests for a previous divide-check exception.

Form:

```
CALL DVCHK ( i )
```

where *i* is an integer variable or array element which will indicate if the divide-check indicator is turned on or off.

Description:

DVCHK tests the divide-check indicator to see if either a fixed- or floating-point divide-check has occurred.

The variable *i* is then set to 1 if the divide-check indicator was on, or it is set to 2 if the indicator was off.

After testing, the divide-check indicator is turned off.

See DIVSET (7.3.3.9).

Example:

```
C      An example of a divide exception is a Fixed Point Divide
C      Exception. This is recognized when the division of a
C      fixed-point number by zero is attempted. Such an
C      exception would occur during execution of the following
C      statements:
      INTEGER DIVISR, DIVDND, TROUBL
      DIVISR = 0
      DIVDND = 8
      QUOTNT = DIVDND/DIVISR
C      This would turn the divide-check indicator on, so the
C      subsequent statement:
      CALL DVCHK(TROUBL)
C      would set variable TROUBL to 1 and turn the indicator
C      off.
```

7.3.3.4. OVERFL

Purpose:

The OVERFL subroutine tests for a previous exponent overflow.

Form:

```
CALL OVERFL ( i )
```

where *i* is an integer variable or array element.

Description:

OVERFL determines whether or not an exponent overflow has occurred since the last call to OVERFL/OVUNFL (or the start of the program, if OVERFL/OVUNFL has not been previously called). The parameter *i* is assigned a value which corresponds to the current overflow status of the program.

If an overflow condition has occurred, the value returned in *i* is 1. An exponent overflow occurs whenever the absolute value of the result of a floating-point addition, subtraction, multiplication, or division is greater than or equal to 2^{127} (approximately 10^{38}) for single precision or 2^{1023} (approximately 10^{307}) for double precision.

If no overflow has occurred, the value returned in *i* is 2.

After the value of *i* has been set, the overflow indicator is cleared.

See OVFSET (7.3.3.8).

Example:

```
A = 1.0E20
```

```
.
```

```
.
```

```
.
```

```
B = A*A
```

```
CALL OVERFL (LAST1)
```

```
C          Assuming that A has not been changed, the value returned  
C          in LAST1 is 1, indicating exponent overflow has occurred.
```

7.3.3.5. UNDRFL

Purpose:

The UNDRFL subroutine tests for a previous exponent underflow.

Form:

```
CALL UNDRFL ( i )
```

where *i* is an integer variable or array element.

Description:

UNDRFL determines whether or not an exponent underflow has occurred since the last call to UNDRFL/OVUNFL (or the start of the program, if UNDRFL/OVUNFL has not been previously called). The parameter *i* is assigned a value which corresponds to the current underflow status of the program.

If an underflow has occurred, the value returned in *i* is 3. An underflow occurs whenever the absolute value of the result of a floating-point addition, subtraction, multiplication, or division is not equal to zero and is less than 2^{-128} (approximately 10^{-38}) for single precision or 2^{-1024} (approximately 10^{-308}) for double precision.

If no underflow has occurred, the value returned in *i* is 2.

After the value of *i* has been set, the underflow indicator is cleared.

See UNDSSET (7.3.3.7).

Example:

```
A = 1.0E-20
```

```
.
```

```
B = A*A
```

```
CALL UNDRFL (LAST1)
```

```
C           Assuming that A has not been changed, the value returned  
C           in LAST1 is 3.
```

7.3.3.6. OVUNFL

Purpose:

- I The OVUNFL subroutine tests for a previous exponent overflow or underflow.

Form:

```
CALL OVUNFL ( i )
```

where *i* is an integer variable or array element.

Description:

OVUNFL determines whether or not an exponent overflow or underflow has occurred since the last call to OVUNFL/OVERFL/UNDRFL (or the start of the program, if OVUNFL/OVERFL/UNDRFL have not been previously called). The parameter *i* is assigned a value which corresponds to the current overflow/underflow status of the program.

If only an overflow condition has occurred, the value returned in *i* is 1. An exponent overflow occurs whenever the absolute value of the result of a floating-point addition, subtraction, multiplication, or division is greater than or equal to 2^{127} (approximately 10^{37}) for single precision or 2^{1023} (approximately 10^{307}) for double precision.

If only an underflow condition has occurred, the value in *i* is 3. An underflow occurs whenever the absolute value of the result of a floating-point addition, subtraction, multiplication or division is not equal to zero and is less than 2^{-128} (approximately 10^{-38}) for single precision or 2^{-1024} (approximately 10^{-308}) for double precision.

If both overflow and underflow have occurred, the value returned in *i* is 4.

If no overflow or underflow has occurred, the value returned in *i* is 2.

After the value of *i* has been set, the overflow/underflow indicators are cleared.

Example:

```
A = 1.0E20
```

```
.
```

```
.
```

```
B = A*A
```

```
CALL OVUNFL (LAST1)
```

C Assuming that A has not been changed, the value returned
C in LAST1 is 1, indicating only overflow has occurred.

7.3.3.7. UNSET

Purpose:

The UNSET subroutine causes subsequent floating-point underflow exceptions to be flagged.

Form:

```
CALL UNSET ( i )
```

where *i* is an integer expression.

Description:

A call to this subroutine allows the next *i* floating-point underflow exceptions to be captured and diagnosed. The following message is generated:

```
WARNING: UNDERFLOW FAULT
```

In addition, if the program had been compiled with the F or C option, the line number and program unit name of the offending statement are printed. After *i* messages, no more will be printed unless another call to UNSET is made. An *i* of zero will stop the capture of these faults.

Checkout debug mode (CZ options) receive an initial automatic default call to UNSET with a count of 20.

Example:

```
@FTN,SIC UND
FTN 10R1 04/01/81-12:44(,0)
1.      COMMON A,B
2.      PRINT *, 'TEST THE UNSET ROUTINE'
3.      A = 1.0E-20
4.      CALL UNSET(2)
5.      B = A*A
6.      PRINT *, 'ONCE'
7.      B = A*A
8.      PRINT *, 'TWICE'
9.      B = A*A
10.     PRINT *, 'THREE TIMES'
11.     END
```

```
END FTN 28 IBANK 45 DBANK 2 COMMON
```

```
ENTERING USER PROGRAM
TEST THE UNSET ROUTINE
```

```
WARNING: UNDERFLOW FAULT
AT LN.      5 OF MAIN PROGRAM
ONCE
```

```
WARNING: UNDERFLOW FAULT
AT LN.      7 OF MAIN PROGRAM
TWICE
THREE TIMES
END PROGRAM EXECUTION
```

7.3.3.8. OVFSET

Purpose:

| The OVFSET subroutine causes subsequent floating-point overflow exceptions to be flagged.

Form:

```
CALL OVFSET( i )
```

where *i* is an integer expression.

Description:

A call to this subroutine causes the next *i* floating-point overflows to be captured. The message:

```
WARNING: OVERFLOW FAULT
```

is printed. In addition, if the program had been compiled with the F or C option, the line number and program unit name of the offending statement are printed.

After *i* messages, the next overflow fault will cause the program to abort, unless another call to OVFSET is done first.

A negative *i* will cause an abort on the first occurrence.

An *i* of zero will stop the capture of these faults. Checkout debug runs (CZ options) automatically get an initial default call to OVFSET with a count of 20.

Example:

```
| @FTN,SIC OVF
| FTN 10R1 04/01/81-13:24(,0)
| 1. COMMON A,B
| 2. PRINT *, 'TEST THE OVFSET ROUTINE'
| 3. A = 1.0E20
| 4. CALL OVFSET(2)
| 5. B = A*A
| 6. PRINT *, 'ONCE'
| 7. B = A*A
| 8. PRINT *, 'TWICE'
| 9. B = A*A
| 10. PRINT *, 'THREE TIMES'
| 11. END

END FTN 28 IBANK 45 DBANK 2 COMMON
```

```
ENTERING USER PROGRAM
TEST THE OVFSET ROUTINE
```

```
WARNING: OVERFLOW FAULT
AT LN. 5 OF MAIN PROGRAM
ONCE
```

WARNING: OVERFLOW FAULT
AT LN. 7 OF MAIN PROGRAM
TWICE

WARNING: OVERFLOW FAULT
ARITHMETIC EXCEPTION COUNT EXPIRED - PROGRAM ABORTED
AT LN. 9 OF MAIN PROGRAM
ERR MODE ERR-TYPE: 00 ERR-CODE: 00
ERROR ADDRESS: 032073 BDI: 300017

7.3.3.9. DIVSET

Purpose:

The DIVSET subroutine causes subsequent divide fault exceptions to be flagged.

Form:

CALL DIVSET(*i*)

where *i* is an integer expression.

Description:

A call to this subroutine causes the next *i* divide checks to be captured. The message:

WARNING: DIVIDE FAULT

is printed. In addition, if the program had been compiled with the F or C option, the line number and program unit name of the offending statement are printed.

After *i* messages, the next divide fault will cause the program to abort, unless another call to DIVSET is done first.

A negative *i* will cause an abort on the first occurrence.

An *i* of zero will stop the capture of these faults.

Checkout debug runs (CZ options) automatically get an initial default call to DIVSET with a count of 20.

Example:

```
@FTN, SIC DIV
FTN 10R1 04/01/81-13:58(,0)
1. COMMON Q, DD, DR
2. PRINT *, 'TEST THE DIVSET ROUTINE'
3. DR = 0.
4. DD = 8.
5. CALL DIVSET(2)
6. Q = DD/DR
7. PRINT *, 'ONCE'
8. Q = DD/DR
9. PRINT *, 'TWICE'
```

```
10.      Q = DD/DR
11.      PRINT *, 'THREE TIMES'
12.      END
```

END FTN 29 IBANK 45 DBANK 3 COMMON

ENTERING USER PROGRAM
TEST THE DIVSET ROUTINE

WARNING: DIVIDE FAULT
AT LN. 6 OF MAIN PROGRAM
ONCE

WARNING: DIVIDE FAULT
AT LN. 8 OF MAIN PROGRAM
TWICE

WARNING: DIVIDE FAULT
ARITHMETIC EXCEPTION COUNT EXPIRED - PROGRAM ABORTED
AT LN. 10 OF MAIN PROGRAM
ERR MODE ERR-TYPE: 00 ERR-CODE: 00
ERROR ADDRESS: 032074 BDI: 300017

7.3.3.10. CMLSET

Purpose:

The CMLSET subroutine causes subsequent errors occurring in the Common Mathematical Library (CML) to be flagged. This includes the mathematical intrinsic functions described in Table 7-2, all exponentiation functions, and complex division.

Form:

CALL CMLSET(*i*)

where *i* is an integer expression.

Description:

A call to this subroutine causes the next *i* CML errors to be diagnosed and control returned back to the point following the CML function call. A function result of zero is returned from the intrinsic function for the error case.

After *i* messages, the next CML error will cause the program to abort, unless another call to CMLSET is done first.

A negative or zero *i* will cause an abort on the first CML error.

Example:

```
@FTN, SC CML
FTN 10R1 10/21/80-12:50(6,)
  1.  *
  2.  *   THIS PROGRAM ALLOWS 2 COMMON MATHEMATICAL LIBRARY
  3.  *   ERRORS TO OCCUR WITHOUT THE PROGRAM BEING ABORTED.
  4.  *   ON THE 3RD MATH ERROR, THE PROGRAM IS ABORTED.
  5.  *
  6.  *
  7.    REAL A(5)
  8.    DATA A/25., -25., -16., -4., 4./
  9.    CALL CMLSET(2)    @ALLOW 2 CML ERRORS BEFORE ABORT
 10.  *
 11.    DO 10 I = 1,5
1 12.      R = SQRT (A(I))
1 13.      WRITE (6,901) A(I), R
1 14.    10  CONTINUE
1 15.  *
16.  901  FORMAT ('SQRT OF', F5.1, ' IS', F12.3)
17.    END
```

END FTN 23 IBANK 37 DBANK

ENTERING USER PROGRAM
SQRT OF 25.0 IS 5.000

ERROR CONDITION IN SQRT ROUTINE CAUSED BY
ARGUMENT UNNORMALIZED OR OUTSIDE ALLOWABLE RANGE
ARG1 = -25.000000
ARG1 OCTAL 57215777777
SQRT REFERENCED AT ABSOLUTE ADDRESS 120304 BDI 300020
AT LN. 12 OF MAIN PROGRAM
SQRT OF -25.0 IS .000

ERROR CONDITION IN SQRT ROUTINE CAUSED BY
ARGUMENT UNNORMALIZED OR OUTSIDE ALLOWABLE RANGE
ARG1 = -16.000000
ARG1 OCTAL 57237777777
SQRT REFERENCED AT ABSOLUTE ADDRESS 120304 BDI 300020
AT LN. 12 OF MAIN PROGRAM
SQRT OF -16.0 IS .000

ERROR CONDITION IN SQRT ROUTINE CAUSED BY
ARGUMENT UNNORMALIZED OR OUTSIDE ALLOWABLE RANGE
ARG1 = -4.000000
ARG1 OCTAL 57437777777
SQRT REFERENCED AT ABSOLUTE ADDRESS 120304 BDI 300020
THIS ADDRESS IS AT LN. 12 OF MAIN PROGRAM
ER EABT\$ ABORT ADR: 104754 BDI: 200020
PROGRAM INITIATED INTERRUPT: EABT\$

7.3.3.11. CHKSV\$ and CHKRS\$

Purpose:

The CHKSV\$ and CHKRS\$ subroutines perform the SAVE and RESTORE checkout debug commands.

Form:

CALL CHKSV\$ [(*i*)] @ does a SAVE command

CALL CHKRS\$ [(*i*)] @ does a RESTORE command

where *i* is an integer expression whose value is > 0.

Description:

These subroutines are available in checkout mode to do a SAVE or RESTORE command (see 10.6.3.12 and 10.6.3.11). The optional integer argument *i* would be the version number.

These subroutines would be useful in making up instructional programs, or games for demonstration. The user does not then need to use the full-debug mode of the checkout feature in order to be able to do a SAVE command for FTNR to work on. Also, in games or instructional programs, the interactive debug facilities get in the way, and a full compilation to execute the program is wasteful and unnecessary. With these two subroutines, it would be possible to have whole libraries of programs which could easily switch back and forth among themselves.

NOTE: In any use of FTNR in restoring a previously run program to execution, the user should not have opened any nonsymbiotic files before processing the corresponding SAVE command.

7.3.3.12. SSWTCH

Purpose:

The SSWTCH subroutine tests one of 12 run condition switches to determine if it is set or not.

Form:

CALL SSWTCH (*i* , *j*)

where:

i is an integer expression indicating which of the 12 switches is to be tested.

j is the integer variable or array element set equal to 1 or 2 if the tested switch was set or not set, respectively.

Description:

SSWTCH regards the middle 12 bits (13-24) of the run condition word as 12 sense switches. Bit 24 corresponds to sense switch 1, bit 23 to sense switch 2, etc. The sense switches can be altered by means of the @SETC Executive control statement (see the EXEC Programmer Reference, UP-4144.2 (see Preface)).

If i is 1 through 12 the i^{th} sense switch is tested. If the i^{th} sense switch is set (one), then j is set equal to 1. If it is not set, then j is set equal to 2.

If $i < 0$ or $i > 12$, an error message is printed and program execution is terminated.

Examples:

```
      CALL SSWTCH(3,JFLAG)
C           Assuming the control statement @SETC 0004 is issued
C           before execution, the variable JFLAG will be set to
C           1 indicating sense switch 3 (bit 22 in the run condition
C           word) is set.
```

```
      I = 12
      CALL SSWTCH(I,JFLAG)
C           Assuming the control statement @SETC 0717 is issued
C           before execution, the variable JFLAG will be set to 2
C           indicating sense switch 12 (bit 13 in the run condition
C           word) is not set.
```

7.3.3.13. SLITE

Purpose:

The SLITE subroutine sets one, or resets all, of the six sense lights which are the sixths of the word labeled F2SLT\$.

Form:

```
CALL SLITE( i )
```

where i is an integer expression indicating which of the six sense lights is to be set.

Description:

The word F2SLT\$ can be used by the programmer to set flags for conditions of his invention. These flags can then be tested with the SLITET function (see 7.3.3.14).

If i is 0, all sense lights are reset to zero. If i is 1 through 6, the i^{th} sense light will be set to one.

If $i < 0$ or $i > 6$, an error message is printed and program execution is terminated.

Examples:

```
      CALL SLITE(0)
C           All six sense lights are reset to zero.
```

```
      CALL SLITE(3)
C           The third sense light (S3 of F2SLT$) is set to 1.
```

7.3.3.14. SLITET

Purpose:

The SLITET subroutine tests and resets one of the six sense lights which are the sixths of the word labeled F2SLT\$.

Form:

```
CALL SLITET ( i , j )
```

where:

- i* is an integer expression indicating which of the six sense lights is to be tested.
- j* is an integer variable or array element set equal to 1 or 2 if the tested light is set or reset respectively.

Description:

If the value of *i* is from 1 through 6, the *i*th sense light is tested. If the *i*th sense light is set (value of one), then *j* is set to 1 and the sense light is reset to 0. If the *i*th sense light is not set (zero), then *j* is set equal to 2.

If $i < 0$ or $i > 6$, an error message is printed and program execution is terminated.

Example:

```
CALL SLITET(3,JFLAG)
C           Assuming sense light 3 is set, JFLAG would be set to 1
C           and sense light 3 would be reset to 0. Assuming
C           sense light 3 is not set, JFLAG would be set to 2.
```

7.3.3.15. EXIT

Purpose:

EXIT terminates execution.

Form:

```
CALL EXIT
```

Description:

The EXIT service subroutine allows programmers to terminate the execution of their programs.

The EXIT subroutine serves the same purpose as the STOP statement (see 4.8). It is provided primarily for compatibility with previous FORTRAN systems.

7.3.3.16. ERTRAN

The ERTRAN subroutines provide access to several Executive Request (ER) functions. (For details on ER functions, see the EXEC Programmer Reference, UP-4144.2 (see Preface).)

7.3.3.16.1. Input/Output Executive Requests

Purpose:

ERTRAN allows the FORTRAN programmer to use some of the input/output Executive Request functions. These are: IO\$ (requests an operation on an input/output device and returns immediately), IOW\$ (requests an operation on an input/output device and returns upon completion of the operation), IOI\$ (identical to IO\$ except that when the operation has completed, an interrupt activity is initiated), IOWI\$ (combines the features of IOI\$ and IOW\$ - control is returned upon completion of input/output and a specified interrupt activity is initiated), IOXI\$ (identical to IOI\$ except that the activity making the request exits), WAIT\$ (delays execution until the input/output operation controlled by a specified I/O packet has been completed), WANY\$ (delays execution until any current input/output operation is completed), TSWAP\$ (closes the current reel for a tape file and requests loading of the next reel of the file), UNLCK\$ (enables an input/output interrupt activity to reduce its switching priority to the priority of the activity which initiated the input/output request), and SYMB\$ (is a packet-driven ER providing a means to request certain symbiont ER functions, character transfer, and some expanded READ\$ capability).

Form:

$$\text{CALL } \left\{ \begin{array}{l} \textit{sub} \\ \textit{sub}_1 (\textit{pkt}) \\ \textit{sub}_2 (\textit{pkt} [, \textit{label}]) \\ \textit{sub}_3 (\textit{pkt}, \textit{label}_1, \textit{label}_2) \end{array} \right\}$$

where:

sub is FWANY or FUNLCK.

*sub*₁ is FIO, FIOI, FIOWI, FIOXI, FTSWAP, or FSYMB.

*sub*₂ is FIOW or FWST.

*sub*₃ is FSTAT.

pkt is an 8-word or 10-word table filled by the user, with input/output packet information.

label is an optional error return.

*label*₁ identifies the return to be used if the input/output operation is still pending.

*label*₂ identifies the return to be used if an input/output operation terminated abnormally.

Description:

Two basic steps are necessary when using these routines:

- Construct an Executive input/output packet (*pkt*) exactly as described in the EXEC Programmer Reference, UP-4144.2 (see Preface). There must be one 8-word table declared for each file to be used simultaneously. For FSYMB the table may be 8 or 10 words.
- Call the required FIO routine, passing the I/O packet as the first argument of the call.

The Executive Request is done with register A0 pointing to the program-constructed input/output packet. This provides the user with full control over the input/output operation and complete error analysis capability.

There are no buffers external to these routines. They are all reentrant at both the program and the activity level.

In order to facilitate creation and manipulation of the input/output packet, a FORTRAN procedure is included in the ASCII FORTRAN library file. This procedure contains statement function and PARAMETER declarations as detailed after the description of the functions which follow. The procedure is made available by the statement:

```
INCLUDE libfile. FIOP
```

where *libfile* is the file name of the FORTRAN run-time library file. The file name is site-dependent.

Note that this input/output mechanism is entirely independent of standard FORTRAN file control. Therefore, no file control tables are built by the FORTRAN system for FIO files.

If FIO is called, the parameter *pkt* which points to the input/output packet is picked up, the Executive Request function IO\$ is referenced, and control is returned immediately to the caller.

If FIOW is called, the parameter *pkt* which points to the input/output packet is picked up, the Executive Request function IOW\$ is referenced and it waits for completion before returning to the caller. If the calling syntax did not include the optional error return, the user should check the status area of the input/output packet by using the ISTAT statement function as provided in FIOP. If the calling syntax did include the error return option, then all abnormal statuses will cause transfer to that label.

If FIOI is called, the parameter *pkt* which points to the input/output packet is picked up, the Executive Request function IOI\$ is referenced, and control is returned immediately to the caller. Upon completion of the input/output operation, the interrupt activity specified in the packet is initiated.

If FIOWI is called, the parameter *pkt* which points to the input/output packet is picked up and the Executive Request function IOWI\$ is referenced. Upon completion of the operation, control is returned to the caller and a specified interrupt activity is initiated.

If FIOXI is called, the parameter *pkt* which points to the input/output packet is picked up, the Executive Request function IOXI\$ is referenced, and the calling activity is terminated. Upon completion of the input/output operation, a specified interrupt activity is initiated.

If FWST is called, the parameter *pkt* which points to the input/output packet is picked up, the Executive Request function WAIT\$ is referenced, and a wait is done if input/output is outstanding on the packet. Upon completion, a status check is made. In the case of an abnormal completion status, a return is done to *label*, if present. If the calling syntax did not include the error return option, control is returned to the instruction following the CALL.

If FWANY is called, the Executive Request function WANY\$ is referenced and a wait is done for completion of any input/output. Upon completion, control is returned to the caller.

If FTSWAP is called, the parameter *pkt* which points to the input/output packet is picked up and the Executive Request function TSWAP\$ is referenced.

If FUNLCK is called, the Executive Request function UNLCK\$ is referenced to enable an input/output interrupt activity which reduces its switching priority to the priority of the activity which initiated the input/output request. Control is then returned to the caller.

If FSYMB is called, the parameter *pkt* pointing to an 8- or 10-word packet is picked up, the Executive Request function SYMB\$ is referenced and control is returned to the caller.

An additional routine, which does not refer to an Executive Request, is FSTAT. If FSTAT is called, the parameter *pkt* which points to the input/output packet is picked up and a check is done on the status of the input/output operation. If the input/output operation is still pending, a return is made to *label*₁. If the input/output operation terminated abnormally, a return is made to *label*₂. If the input/output operation terminated normally, a normal return is taken.

The information required in the Executive Request packet is placed in the various fields by use of statement function and PARAMETER statements in the FORTRAN procedure FIOP. This FORTRAN procedure must be included with the FORTRAN source program (see 8.1). There are statement function names for all of the fields in the Executive Request packet except the first two words, which contain the Fielddata internal file name or file reference number left-justified and space-filled. These two words can be initialized with the DATA statement using the Fielddata representation of either the internal file name or the file reference number. This is most efficient since it is done at compile time. However, an alternate method would be the usage of the FASCFD run-time conversion routine (see 7.3.3.19).

The following list of statement function names represents the various fields of the Executive Request packet, as described in the EXEC Programmer Reference, UP-4 144.2 (see Preface). Each is contained in FIOP. Each statement function reference requires one argument which is the input/output packet name.

<u>Field</u>	<u>Description</u>
ACTID	<i>int-act-id</i> (numeric identity used to identify the interrupt activity)
ACTADDR	<i>interrupt-activity-addr</i> (interrupt activity address)
ISTAT	<i>status</i> (status of the last function performed)
FUNCT	<i>function</i> (denotes function to be performed)
AFC	AFC (abnormal frame count for tape I/O)
SUBST	<i>final-word-count-returned-by-I/O</i>
GINC	G (incrementation flag)

NWRDS	<i>word-count</i> (number of words to be transferred)
BUFAD	<i>buffer-addr</i> (memory address for transfer - use LOC intrinsic function)
TRKAD	<i>mass-storage-addr</i> (relative track/sector address for start of I/O)
SRCHS	<i>search-sentinel</i>
SFDA	<i>search-find-drum-addr</i> (search find address)

The following list of statement function names represents the various fields of the Executive Request packet for SYMB\$. Each statement function reference requires one argument which is the input/output packet name.

<u>Field</u>	<u>Description</u>
IFUNC	<i>function</i> (describes the action to be performed on the specified file)
IMODE	<i>mode</i> (provides further description of the function)
ISTAT	<i>status</i> (status of last function performed)
IERCD	<i>I/O-status</i> (I/O error code if SYMB\$ was terminated because of an I/O error)
ICCN	<i>control-card-index</i> (CLIST index for an image read)
ISUBST	<i>sub-status returned</i> (further information status)
ICHCT	<i>character-count</i> (number of characters to transfer)
IIMGAD	<i>image-address</i> (address to put/get an image)
ITTNF	<i>TT-number-field</i> (which translate table to use)
ICHSZ	<i>character-size-field</i> (6-bit or 9-bit byte)
IFCHCT	<i>final-character-count-transferred</i> (number of words or characters transferred)
ISPCE	<i>spacing</i> (number of lines to space before an image is written)
IEOFS	<i>EOF-sentinel</i> (Fieldata or ASCII character returned in column 6)
IWRCC	<i>character-count</i> (number of characters to transfer)
IWRADD	<i>image-address</i> (address of an image)
IWRFC	<i>final-character-count-transferred</i> (number of characters or words transferred)

The following is a list of names from PARAMETER statements declared in FIOP that define the values for the I/O functions for the FUNCT field. For more information, see the EXEC Programmer Reference, UP-4144.2 (see Preface).

<u>Parameter</u>	<u>Octal</u>	<u>Function</u>
FW	10	Write
FWEF	11	Write end of file
FCW	12	Contingency write
FSW	13	Skip write
FABW	14	TIP recoverable write
FGW	15	Gather write
FACQ	16	Acquire
FABSW	17	Absolute write whole unit/extended acquire
FR	20	Read
FRB	21	Read backward
FRR	22	Read and release
FREL	23	Release
FBRD	24	Block read drum
FRDL	25	Read and lock
FUNL	26	Unlock
FABR	27	Absolute read, TIP recoverable read
FTSA	30	Track search all words
FTSF	31	Track search first word
FPSA	32	Position search all words
FPSF	33	Position search first word
FSD	34	Search drum
FBSD	35	Block search drum
FSRD	36	Search read drum
FBSRD	37	Block search read drum
FREW	40	Rewind
FREWI	41	Rewind with interlock
FSM	42	Set mode
FSCR	43	Scatter read
FSCRB	44	Scatter read backward
FWR	45	Write, then read
FABSR	47	Absolute read whole unit
FMF	50	Move forward
FMB	51	Move backward
FFSF	52	Forward space file
FBSF	53	Backspace file
FCN	55	Mode set

The following is a list of names from PARAMETER statements declared in FIOP defining the values for the mode field for SYMB\$:

<u>Parameter</u>	<u>Octal</u>	<u>Description</u>
IASC	1	ASCII request
ITRUN	2	Truncate an image
IUNTR	4	Untranslate request
IPUALT	10	Punch alternate file
ISPEC	20	Special translation index

The following is a list of the names declared by statement function statements in FIOP of the fields for the set mode function (FORTRAN PARAMETER - FSM):

<u>Field Name</u>	<u>Bits</u>
FSMF1	34 - 35
FSMF2	32 - 33
FSMF3	30 - 31
FSMF4	28 - 29
FSMF5	26 - 27
FSMF6	22 - 25
FSMF7	20 - 21
FSMF8	18 - 19
FSMF9	0 - 17

The fields of the set mode function are set as described in the EXEC Programmer Reference, UP-4144.2 (see Preface).

Examples:

```

SUBROUTINE FASTIO(BUF,NADDR)
C      This subroutine will read 112 words from track NADDR
C      into the array BUF.
LOGICAL INIT/.FALSE./
C      Declare 8-word table for packet.
INTEGER IOT(8)
C      Include the FORTRAN procedure in the program.
C      File name may have to be changed depending on
C      site conventions.
INCLUDE SYS$*FTNRLIB$.FIOP
C      Initialize first two words with file name in Fieldata,
C      words three through eight to zero.
C      A unit reference number could have been used.
DATA IOT(1), IOT(2), (IOT(1), 1 = 3, 8) /'MYFILE'F,'      'F,6*0/
IF (INIT) GO TO 110
INIT = .TRUE.
C      Assign the file.
STAT=FACSF('@ASG,A MYFILE . ')
C      Check that the file assignment is OK.
IF (STAT.LT.0) CALL FABORT
NWRDS(IOT) = 112
C      112 words to be read.
FUNCT(IOT) = FR
110 CONTINUE
BUFAD(IOT) = LOC(BUF)
C      Set file track address for input.
TRKAD(IOT) = NADDR
C      Call to do input/output
CALL FLOW(IOT,$500)
RETURN
C      Following is error handler
500 CONTINUE
PRINT *, 'DID NOT WORK'
CALL FEXIT
RETURN
END
```

The following program shows one way to use FSymb.

```
PROGRAM AWRITE
*       This program will write 132 characters from BUFFER through
*       SYMB$ if the printer allows 132 characters
CHARACTER*204 BUFFER
INTEGER PACKET(10)
*       Include the statement functions and parameters to use SYMB$
INCLUDE F10P
*       Set the file name to the PRINT$ ER code
PACKET(1)=14
PACKET(2)=0
*       Set the function code to PRINT$
INFUNC(PACKET)=FW
*       Set the mode to ASCII
IMODE(PACKET)=IASC
*       Set the number-of-characters transferred
ICHCT(PACKET)=132
*       Set the image address
IIMGAD(PACKET)=LOC(BUFFER)
*       Clear the remainder of the SYMB$ packet
DO 10 I=6,10,1
10      PACKET(I)=0
*       Initialize the 132-character BUFFER
BUFFER='TEST BUFFER LENGTH OF 132 CHARACTERS'
BUFFER(111:132)='END OF 132 CHARACTERS'
BUFFER(194:204)='END OF LINE'
*       Call SYMB$
CALL FSymb(PACKET)
STOP
END
```

7.3.3.16.2. Miscellaneous Executive Requests

Purpose:

These Executive Requests allow the FORTRAN programmer to refer to each of the Executive Request functions: ABORT\$ (abort run), ACSF\$ (generate control statement), ERR\$ (error exit), EXIT\$ (program exit), SETC\$ (set condition word), COND\$ (retrieve condition word), and DATE\$ (request date and time).

Form:

$$\text{CALL } \left\{ \begin{array}{l} r_1 \\ r_2 \text{ (arg)} \\ r_3 \text{ (arg1, arg2)} \end{array} \right\}$$

where:

r_1 is FABORT, FERR, or FEXIT.

r_2 is FACSF, FACSF2, FSETC, or FCOND.

*r*₃ is FDATE or ADATE.

arg is a character array or a character expression for FACS F and FACS F2, an integer expression for FSETC, or an integer variable or array element for FCOND.

*arg*₁, *arg*₂ are variables or array elements.

Description:

If FABORT is called, the Executive Request function ABORT\$ is referred to. All current activities are terminated and the run is terminated in an abort condition immediately; files are not closed.

If FERR is called, the Executive Request function ERR\$ is referred to. Only the activity in error is terminated.

If FEXIT is called, the Executive Request function EXIT\$ is referred to. The routine provides program termination. No file-closing actions are performed if FEXIT is called.

If FACS F is called, the Executive Request function ACSF\$ is referred to. The routine submits an Executive control statement image (*arg*) for interpretation and processing. The image submitted must be a character array or a character expression containing one of the control statements in the list which follows. The control statement must not be longer than 80 characters and must be terminated by the character sequence: blank, period, blank, or a word of blanks. If FACS F is called as a function, FACS F must be typed as integer and then the status resulting from the ACSF\$ call is returned as the value of the function. (See the EXEC Programmer Reference, UP-4 144.2 (see Preface).) A subprogram name (for example, FACS F) can only be used in one way in a given program. Thus, FACS F may not be used in a CALL statement and also be used as a function name in the same program.

Statement

Use

@ADD	Add to runstream
@ASG	Assign a file
@BRKPT	Breakpoint symbiont output files
@CAT	Catalog a tile
@CKPT	Produce checkpoint dump of this run
@FREE	Deassign a file
@LOG	Message to the Master Log File.
@MODE	Set mode and/or noise constant for tape file
@QUAL	File qualification
@RSTRT	Restart run whose checkpoint dump was saved by @CKPT
@START	Schedule an independent run
@SYM	Queue files for symbiont processing
@USE	Associate internal to external file name

FACS F2 is the same as FACS F, except that no ERTRAN error message will be printed if an error status is returned from the ACSF\$ Executive Request. Users must perform their own error checking (using the status returned as the FACS F2 function value) and processing.

If FSETC is called, the Executive Request function SETC\$ is referred to. The subroutine places (sets) the contents of the lower third (bits 25-36) of *arg* in the corresponding third of the run condition word. The lower two thirds of the run condition word are used as a flag which can be either tested by the control statement @TEST or retrieved by the FORTRAN call CALL FCOND(*a*) and then tested.

If FCOND is called, the Executive Request function COND\$ is referred to. The subroutine retrieves the condition word and makes it available to the user in *arg*.

If FDATE is called, the Executive Request function DATE\$ is referred to. The subroutine supplies the user with the current date and time in *arg1* and *arg2*, respectively. The data in *arg1* is the Fieldata character form MMDDYY where MM represents the month (01-12), DD the day (01-31), and YY the last two digits of the year (00-99). The time in *arg2* is in the Fieldata character form HHMMSS where HH represents the hours (00-24), MM the minutes (00-60), and SS the seconds (00-60). The first words of *arg1* and *arg2* (which are both variables or array elements) will be filled with the 6-bit Fieldata characters described previously.

If ADATE is called, the Executive Request function DATE\$ is also referred to as in FDATE, but the date and time are returned in ASCII character form. Variables *arg1* and *arg2* must be character variables or character array elements of eight characters in length. The first six character positions of each variable will be filled with the ASCII character form of the data and time in the format described for FDATE above. The remaining two characters will be space-filled.

Examples:

```
                INTEGER FACSF2
                CHARACTER ASG*20, DATE*8, TIME*8
                DATA ASG / '@ASG,A FILENAM . '/
                CALL FACSF(ASG)
C               This call would attempt to assign the file FILENAM by
C               referring to the Executive Request function ACSF$.
                ISTAT = FACSF2(ASG)
C               This function reference attempts the assignment and
C               puts the result status in ISTAT.
                IF(ISTAT .LT. 0) CALL FERR      @ terminate on error
                CALL ADATE(DATE, TIME)
C               This call supplies the user with the date ('MMDDYY ') in DATE
C               and the time ('HHMMSS ') in TIME.
                PRINT *, DATE, TIME
                END
```

7.3.3.17. NTRAN\$

Purpose:

NTRAN\$ provides a tool for reading or writing binary information on tape or mass storage, and also provides for I/O buffering. NTRAN\$ I/O processing is completely separated from normal FORTRAN I/O processing. NTRAN\$ I/O is only accessible by calls to the NTRAN\$ service subroutine, and normal FORTRAN I/O is accessible only by FORTRAN I/O statements such as READ, WRITE, BACKSPACE, ENDFILE, OPEN, CLOSE, and INQUIRE. AN NTRAN\$-created file cannot be referred to by normal FORTRAN I/O statements.

The ASCII FORTRAN NTRAN\$ subroutine has exactly the same syntax as the FORTRAN V NTRAN subroutine (see the FORTRAN V Library Programmer Reference, UP-7876 (see Preface)). The only difference is that NTRAN in the FORTRAN V call is replaced by NTRAN\$ in the ASCII FORTRAN call.

Form:

```
CALL NTRAN$(unit, sequence-of-operations)
```

where *unit* is an integer expression designating the logical unit, and the *sequence-of-operations* is any list of I/O operations (as specified in 7.3.3.17.1) to be performed in order on the specified unit.

Description:

If the unit is not busy, NTRAN\$ initiates the first operation, stacks the rest in a waiting list, and then returns to the calling program. If the unit is already busy, then the entire sequence is stacked in a waiting list and chained to any previously stacked operations. The exceptions are operations 16 to 22; when they are encountered, NTRAN\$ waits for the completion of all previous operations for that unit before returning to the calling program. When an interrupt occurs, NTRAN\$ records the transmission status, initiates the next operation in the chain, and returns control to the interrupted calling program. Priority tasks are not supported.

The I/O operations provided by NTRAN\$ are as follows:

1. Write (tape or mass storage)
2. Read (tape or mass storage)
3. Block Read (tape or mass storage)
4. Search Read (tape or mass storage)
5. Search Mass Storage
6. Position Mass Storage
7. Position Tape by Block (tape)
8. Position Tape by Files (tape)
9. Write End of File (tape)
10. Rewind (tape or mass storage)
11. Rewind/Interlock (same as Rewind for mass storage)
12. Set Tape Density Medium (tape)
13. Set Tape Density Low (tape)
14. Set Tape Parity Odd (tape)
15. Set Tape Parity Even (tape)
16. Initialize Multireel File (tape)
17. Swap Reels for Multireel File (tape)
18. Reassign Unit (tape or mass storage)
19. Assign Unit to External File Name (tape or mass storage)
20. NOP (tape or mass storage)
21. Get Device
22. Wait and Unstack then Release Unit (tape or mass storage)
23. Set Tape Density High (tape)

In order to use NTRAN\$, a FORTRAN program must have some way to check the status of the transmission. For this reason, every block of main storage which is used for I/O operations has a block status word (an integer variable) associated with it; the name of the status word is specified in the argument list of the CALL.

When NTRAN\$ is called, the list of arguments is searched for status words, and these are all set to a value (-1) which indicates transmission is not complete. When an interrupt occurs, the corresponding status word is set by NTRAN\$ to a value which indicates the nature of completion, whether normal (a positive value indicating the number of words transmitted), abnormal (value = -2) or in error (value = -3 or -4). The status words for each operation are defined in 7.3.3.17.1.

When NTRAN\$ generates -2 or -3, it releases all operations stacked for the unit which have not been started. The offending operation is marked to abort and is left stacked. Any further calls of NTRAN\$, requesting the above described unit (except operation 22) will not be performed or stacked, but will generate a particular status code (-4). Operation 22 may be used to release the abort condition for a unit. This allows the programmer to regain control after trying to read or write past an end-of-file, end-of-drum, or end-of-tape.

An attempt to read or write zero words ($n=0$) will result in the function being ignored.

The following errors will generate a status word of -3:

- Hardware errors
- Parity and character count errors
- Illegal unit specified

NOTE: Legal units are all tapes and mass storage files.

The user should also note that at compilation time, the ASCII FORTRAN compiler will print out a warning each time NTRAN\$ is called from the same program unit with a different number of arguments than was specified on the first call (in the program unit). These warnings can be ignored by the user.

7.3.3.17.1. Operations

An operation is defined in the argument list by a group of arguments. The first argument for an operation identifies the type of operation. It is followed by the parameters for the operation; these are fixed in number and order of occurrence by the type of operation. Several operations may be grouped in a single call to NTRAN\$.

When a mass storage file is referred to, the current mass storage address for that file is the starting address for the file only if the mass storage file was never referred to in the current run. If the mass storage file was referred to before, the current mass storage address is the current address before the last CALL statement using the file plus the number of words transmitted or positioned in that CALL statement. In order to reach the starting address of the file, operation 10 and 22 can be used.

For example, in CALL NTRAN\$ (3, 9, 10, 22):

3 = unit number

9 = end-of-file when operation is completed

10 = rewind unit

22 = all operations on unit must be completed before another function is issued.

The cited example is a stacked operation.

NOTE: For sector-formatted mass storage I/O, the specified mass storage address is a sector count and not a word count as for word-addressable mass storage I/O. However, with normal termination, the status variable associated with a main storage transfer will indicate actual number of words transmitted. It is then up to the user to perform the covered divide with the sector size in order to retrieve the corresponding sector count.

For search operations on sector-formatted mass storage, if a find is made, the mass storage address will point to the sector containing the matching item; a following read function will therefore not necessarily start reading the matched item.

■ Write

The argument group is: $1, n, b, l$

in which n is an integer constant or variable which specifies the block length; b is a variable name from which data is to be written; and l is an integer variable, the status word, which is set by NTRAN\$ as follows:

- 1 = transmission not complete
- 2 = end of the tape or mass storage file
- 3 = device error
- 4 = transmission aborted (previous operation had -2 or -3 status).

If the transmission is completed normally, l receives the number of words transmitted (n).

■ Read

The argument group is: $2, n, b, l$

in which n is an integer constant or variable which specifies the length of the main storage block which will receive the data (for tape, n is the maximum number of words which will be transmitted from the tape block; for mass storage, n words will be transmitted), b is a variable which is the name of a main storage area into which the data is to be read, and l is an integer variable (the status word), which is to be set as follows:

- 1 = transmission not complete
- 2 = end of file (no words read from mass storage)
- 3 = device error
- 4 = transmission aborted (previous operation had -2 or -3 status).

If the transmission is completed normally, l receives the number of words transmitted (n).

■ Block Read

The argument group is: $3, n, b, l$

A block read for tape and sector-formatted mass storage is the same as an ordinary read. For word-addressable mass storage, transmission is terminated by reading a word of all 1 bits (called end-of-block word). n is the maximum number of words which can be transmitted. l (the status variable) receives the actual number of words transmitted if the operation is completed normally; otherwise l is set as in READ. b has the same definition as in read.

■ Search Read

The argument group is: $4, s, n, b, l$

in which s (a sentinel word) is a constant or a variable which is used in searching tape or mass storage.

For tape, the first word of each block is compared to the sentinel and, when a match is found, that block (including the sentinel word) is read. For mass storage, starting at the current mass storage address, each word is compared to the sentinel until a match is found or until all remaining words of the granule (track or position) are tested. An unsuccessful search results in status (-2) for l . If no find was made, the user may request additional searches by setting the mass storage address of a different granule (track or position). For sector-formatted mass storage, a track search is employed; if no find is made, the user may request additional searches.

When a match is found on word-addressable mass storage, the block (n words) is read into b . When a match is found on a sector-formatted mass storage, the entire sector containing the matched sentinel will be read into b .

■ Search Mass Storage

The argument group is: $5,s$

in which s is a constant or variable sentinel word. Starting at the current mass storage address, each word is compared to the sentinel until a match is found or until all remaining words of the granule (track or position) are tested. The mass storage address of the match becomes the new current mass storage address (the first mass storage address to be read or written is that of the matched mass storage address). When a match is found on a sector-formatted mass storage device, the mass storage address will point to the sector containing the matched sentinel. If a match has not been made, the address does not change.

■ Position Mass Storage

The argument group is: $6,n$

in which n is an integer constant or variable, positive or negative, which is added to the current mass storage address to form a new current mass storage address. If n is negative and the current mass storage address plus n is less than the starting address of the mass storage file, the current mass storage address is set to the starting address of the mass storage file. N is the word count for word-addressable mass storage, and the sector count for sector-formatted mass storage.

■ Position Tape By Blocks

The argument group is: $7,n$

in which n is an integer constant or variable which specifies the number of blocks to space over on tape. Positive n for forward spacing; negative n for backspacing.

■ Position Tape By Files

The argument group is: $8,n$

in which n is an integer constant or variable which specified the number of file marks to space over. Positive n for forward spacing; negative n for backspacing. The operation is terminated by moving over the n th file mark, by reaching the load point (back spacing), or by reaching the end of tape (forward spacing).

■ End File

The argument group is: 9

For tape, an end-of-file mark is written.

■ Rewind

The argument group is: 10

■ Rewind/Interlock

The argument group is: 11

For tape, a rewind/interlock is given. For word-addressable mass storage and sector-formatted mass storage, the operation is the same as a rewind.

NOTE: The following four operations pertain to magnetic tape density and parity setting (available only on UNISERVO 7-track tape units). If not specified, the setting will be system standard.

■ Set Tape Density Medium (556 bpi)

The argument group is: 12

■ Set Tape Density Low (200 bpi)

The argument group is: 13

■ Set Tape Parity Odd (binary standard)

The argument group is: 14

■ Set Tape Parity Even (BCD standard)

The argument group is: 15

NOTE: Density and parity setting routines set density and parity for all tape units tied to a logical unit when doing multireel processing.

■ Initialize Multireel File

The argument group is: 16

The operation is used to reinitialize the cycle of tape swapping when more than one pass is made over a multireel file.

■ Swap Reels

The argument group is: 17

This operation is used to access the next physical unit in a multireel file. The old physical unit is not rewound. Any number of physical units may be assigned to a given unit, by using a sequence of operation 17's.

■ Deassign Unit

The argument group is: 18

This operation releases a unit, without rewinding the physical units (frees internal file name, unit, from external file name).

■ Assign Unit to External File Name

The argument group is: 19,*x*,*l*

This operation assigns a unit to an external file name, or links an internal name to an already assigned file, where *x* is an array containing a two-word file name in Fielddata and *l* is an integer variable which is set to a value which indicates the following:

0 = assignment made

1 = error in call statement, no assignment made.

■ NCP For Compatibility (old function dropped)

The argument group is: 20,*l*

■ Retrieve Device

The argument group is: 21,*x*

This operation retrieves the device code for the specified unit, in which *x* is the variable which receives the device code upon return.

■ Wait and Unstack

The argument group is: 22

This operation causes a wait in NTRAN\$ until all previous operations, for the specified logical unit, are complete before stacking any further operations or returning to the user's program. It also removes any operation which has caused an abnormal or error status which is still stacked against the unit specified.

■ Set Tape Density High (800 bpi - not compatible)

The argument group is: 23

7.3.3.17.2. NTRAN\$ Error Messages

NTRAN\$ will, under certain error conditions, produce an error message. The unit number will be identified within the message. The FORTRAN program and the line number where the call to NTRAN\$ was made will be identified under the error message.

The eight possible error messages produced by NTRAN\$ are:

■ ****NTRAN ERROR* UNIT *n*: NO PACKET SPACE AVAILABLE**

This message indicates that all available NTRAN\$ packets are in use and that another packet is requested.

Suggested Action: Reassemble the ASCII FORTRAN library element F2NTRAN\$ and increase by the number-of-packets parameter NPKTS.

■ ****NTRAN ERROR* UNIT *n* IS NOT AVAILABLE FOR NTRAN.**

A reference to a unit was made that is already in use by normal FORTRAN I/O processing.

Suggested Action: Change unit number.

■ ****NTRAN ERROR* UNIT *n* NOT ASSIGNED.**

A reference to an unassigned unit was made with a function other than write function 1 (see 7.3.3.17) or assign function 19.

Suggested Action: If a write function 1 had been used as the first reference, a dynamic assign of a mass storage file (scratch) would have been made. If a scratch file was not intended, an assignment has to be made either by assign function 19 or by an assign card.

■ ****NTRAN ERROR* UNIT *n* HAS IMPROPER DEVICE**

Requested function is not available for the device assigned to this unit. The requested function will be ignored.

Suggested Action: If action is wanted for the requested function, a unit with another device assigned has to be used.

■ ****NTRAN ERROR* UNIT *n* HAS ILLEGAL FUNCTION CODE**

■ ****NTRAN ERROR* UNIT *n*: NUMBER OF ARGUMENTS IN STACK EXCEEDS TABLE LENGTH.**

This message indicates that the number of arguments in call is greater than the maximum calling sequence table length.

Suggested Action: Reassemble ASCII FORTRAN library element F2NTRAN\$ and increase the NCT length (NCTLT).

■ ****NTRAN ERROR* UNIT *n*: SYNTAX ERROR FOR FILE NAME**

This message indicates an illegal character used in the file name for NTRAN\$ function 19 (see 7.3.3.17).

■ ****NTRAN WARNING* BANKED ARGUMENTS ARE NOT ALLOWED**

Suggested Action: Do not pass banked data items to NTRAN\$.

NOTE: *The user must not change any argument of an argument group before the function is completed, that is, before the status word (if any) has been changed from -1 to another value. All NTRAN\$ functions are executed in sequence; the completion of one function implies completion, successful or unsuccessful, of all preceding functions.*

7.3.3.18. CLOSE

Purpose:

The CLOSE subroutine closes a FORTRAN data file and frees all main storage associated with the file such as the file control table and buffer areas.

Form:

```
CALL CLOSE ( i, j )
```

where *i* and *j* are integer expressions.

Description:

This subroutine will close the file associated with the unit reference number designated by the first parameter.

The file will be rewound if the second argument is nonzero (for tape only).

Examples:

```
      I = 3
      J = 0
      CALL CLOSE (I,J)
C           The file associated with unit number 3 will be closed
C           and no rewind of the tape file will occur.
```

7.3.3.19. FASCFD and FFDASC

Purpose:

The FASCFD and FFDASC subroutines provide FORTRAN interface to the FDASC conversion routine. FDASC allows for conversion from ASCII to Fielddata and from Fielddata to ASCII.

Form:

```
CALL r ( i, a, b )
```

where:

- r* is FASCFD or FFDASC.
- i* is a positive integer variable or array element specifying the number of words to be converted. Upon return from the FASCFD/FFDASC call, *i* will contain the length in words of the converted string.
- a* is the source (an expression) that is to be converted.
- b* is the target (variable or array element) for the converted characters.

Description:

In the following descriptions, ASCII characters are assumed to be packed four characters per word and Fielddata characters are assumed to be packed six characters per word.

If FASCFD is called, the $4i$ ASCII characters in a are converted to Fielddata characters and stored in b . If $4i/6$ has a remainder R other than zero, then $6-R$ Fielddata spaces will follow the last character in b , space-filling the last word. In addition, if the last word of b is all spaces, the word count returned in i will not reflect this word. ASCII characters are converted as described in the SYSLIB Programmer Reference, UP-8728 (see Preface).

If FFDASC is called, the $6i$ Fielddata characters in a are converted to ASCII characters and stored in b . If $6i/4$ has a remainder of R other than zero, then $4-R$ ASCII spaces will follow the last character in b , space-filling the last word. If the last word of b is all spaces, the word count returned in i will not reflect this word. In this case, a conversion from Fielddata to ASCII will not be performed correctly.

For either call, if a and b are the same character variable then the source characters will be destroyed.

The a and b variables must be in the same data bank (D-bank).

Example:

```
IWC = 13
CALL FASCFD (IWC,ASC,FD)
C           The first 52 ASCII characters in ASC are converted to
C           Fielddata and stored in FD. Upon return, IWC equals 9.
C           If the last four characters (49-52) of ASC were spaces,
C           then IWC would be set to 8 upon return.
```

The following call to FASCFD could result in a serious error:

```
CALL FASCFD(2,ASC,FD)
```

Even though a constant was passed, the value of the constant will be changed upon return from FASCFD. Since the compiler assumes that the values of constants never change and reuses them throughout a compilation, logic errors could easily result. Note that this situation can happen any time a constant is passed as an argument to a routine which changes the value of the corresponding formal argument.

7.3.3.20. MAXAD\$

Purpose:

The MAXAD\$ subroutine changes one or more items in the Common Storage Management System (CSMS) packet, and returns a status. CSMS is used mainly for I/O buffers.

Form:

CALL MAXAD\$ (*max*, *mcore*, *lcore*)

where:

max is the maximum address that the program can reach (that is, CSMS will never request main storage past this address). If *max* is -1 or 0, the default 0777777 (decimal value 262143) is assumed.

mcore is the request main storage increment size (that is, the minimum amount of main storage obtained each time an ER MCORE\$ is done). If *mcore* is -1, the default 010000 (decimal value 4096) is assumed. If *mcore* is 0, no requests for main storage will be made.

lcore is the minimum amount of freed storage that will remain in the allocated area after an ER LCORE\$ is performed by the CSMS (to release storage that has been freed). If *lcore* is -1, the default 020000 (decimal value 8192) is assumed. If *lcore* is 0, no release of main storage will be done. It is recommended that if *lcore* is specified and is nonzero, it should be at least twice as large as the request for main storage increment size *mcore*.

Description:

This service subroutine may not be called from checkout mode, as checkout does not use the CSMS feature.

If any of the three parameters is passed as -2, this subroutine will not change the corresponding packet location.

This function returns 0 or -1, which indicates good or bad status, respectively. This return status may be tested by the user, if MAXAD\$ was referenced as a function (rather than as a subroutine). A zero status indicates no errors. A bad status (-1) indicates that one of the following has occurred:

- One of the parameters was greater than 0777777 (decimal value 262143). The parameter is ignored.
- The current maximum address in the packet (that is, the current end of the control bank) was greater than *max* (first parameter). In this case, the absolute maximum address in the packet (that is, the address that the program will never exceed) is set to the current maximum address.

If the request for main storage increment size in the packet is already zero when this routine is entered, then no action is performed by this routine. This is because an initial reserve was probably specified in F2FCA (see G.7), and the CSMS routines use that main storage with no ER MCORE\$ or ER LCORE\$. Note that a good status (zero) will be returned in this case.

Note that caution must be used when specifying *mcore* = 0, or when specifying *max* as anything but the default. CSMS will error terminate in certain cases.

7.3.3.21. LOC

Purpose:

The LOC integer function returns the address of its argument.

Form:

LOC(*name*)

where *name* is the argument whose address is returned as the function value.

Description:

Note that LOC should be typed as integer in the calling routine, if an IMPLICIT statement types the letter L as non-integer, since LOC is in the service subroutine class and does not have an inherent type.

7.3.3.22. MCORF\$ and LCORF\$

Purpose:

The MCORF\$ and LCORF\$ service routines give a primitive entry into the ASCII FORTRAN storage allocator, so that dynamic pseudo-arrays may be allocated and released.

Form:

iadr=MCORF\$(*isize*) @ Function call

CALL LCORF\$(*iadr*)

where:

iadr is the address of the buffer to be freed (*iadr* was returned as the corresponding MCORF\$ function result).

isize is the number of words desired in the dynamic pseudo-array (passed to function MCORF\$, which returns an address *iadr*).

Description:

Since it is difficult to use an address in the FORTRAN language, base-offset type referencing must be performed in order to use the address returned by MCORF\$.

The pseudo-arrays created by this process do not have all of the attributes of a normal FORTRAN array. For example, the pseudo-array names (D1 and D2 in the first example below) with no following subscripts cannot be passed to a subprogram or to I/O (since they are actually statement functions which require an argument). However, individual array elements of the pseudo-arrays can appear anywhere.

Examples:

The following example uses a dummy array, DUMMY, and statement functions to create two dynamic pseudo-arrays, D1 and D2 (which are both single precision real).

```
      DIMENSION DUMMY(1)
      DEFINE D1( I ) = DUMMY( I + ID1OFF )
      DEFINE D2( I ) = DUMMY( I + ID2OFF )
      ID1OFF = 2000      @ MAKE D1 2000 ELEMENTS
      ID2OFF = 3000      @ MAKE D2 3000 ELEMENTS
      CALL GET( DUMMY, ID1OFF )
      CALL GET( DUMMY, ID2OFF )

C
C - INITIALIZE ARRAYS
C
      DO 10 J = 1, 2000
10         D1( J ) = 0.
      DO 20 J = 1, 3000
20         D2( J ) = 0.
C
C - NOW USE D1 AND D2 ARRAYS
C

C
C - NOW FREE D1 AND D2.
C - NOTE THAT ID1OFF AND ID2OFF WERE NEVER CHANGED
C
      CALL FREE ( DUMMY, ID1OFF )
      CALL FREE ( DUMMY, ID2OFF )
      END

      SUBROUTINE GET ( DUM, ISZ )
      ISZ = MCOF$ ( ISZ ) - LOC ( DUM )
      END

      SUBROUTINE FREE ( DUM, ISZ )
      CALL LCOF$ ( ISZ + LOC ( DUM ) )
      ISZ = 0      @ PREVENTIVE MEDICINE
      END
```

The following example is similar to the preceding example, except that it creates a pseudo-array, ADU(1000), with two words per element (for example, double precision real).

```
PARAMETER (IMAXX = 1000) @ Dimension of pseudo-array
DOUBLE PRECISION AD(1), ADU
INTEGER ADI
DEFINE ADU(IX) = AD(IX+ADI)
ADI = IMAXX @ Cleared by free routine
CALL GET2(AD,ADI) @ Get storage for array ADU

        USE ARRAY ADU

CALL FREE2(AD, ADI) @ Free storage for ADU
END

SUBROUTINE GET2(D, IDIM)
*
      DOUBLE PRECISION D
* FOR 2 WORDS/ELEMENT ARRAY
*
      COMMON /STRG2$/INUMD, ISVDA(40), ISVDX(40)
      DATA INUMD/0/
      IA=MCORF$(IDIM*2+3)
      IDIM=(IA-LOC(D))/2 + 1
      IX = IDIM*2+LOC(D) @ APPROX ADDR
      IF (INUMD.GE.40) STOP 'GET2 MAX'
      INUMD = INUMD+1
      ISVDA(INUMD)=IA @ ACTUAL ADDR OBTAINED
      ISVDX(INUMD)=IX @ APPROX ADDR
      END

SUBROUTINE FREE2(D, IDIM)
DOUBLE PRECISION D
COMMON /STRG2$/INUMD, ISVDA(40), ISVDX(40)
*
* FIRST CALC THE APPROX. ADDRESS, THEN SEARCH FOR THE ACTUAL ADDRESS
      IX=IDIM*2+LOC(D)
      DO 10 I=1, INUMD
      IF (ISVDX(I).EQ.IX) THEN
          CALL LCORF$(ISVDA(I))
          IDIM = 0
          INUMD=INUMD-1
          IF (I.EQ.INUMD+1)RETURN
          IF (INUMD.EQ.0)RETURN
          ISVDA(I)=ISVDA(INUMD+1)
          ISVDX(I)=ISVDX(INUMD+1)
          RETURN
      ENDIF
10 CONTINUE
STOP 'FREE2 NO-FIND'
END
```

The following example creates a CHARACTER*3 pseudo-array, A3U (1000).

```
PARAMETER (IMAXX = 1000)
CHARACTER*3 A3(1), A3U
INTEGER A3I
DEFINE A3U(IX) = A3(IX+A3I)
ADI=IMAXX @ Cleared by free routine
CALL GETC3 (A3,A3I) @ Get storage for array A3U
.
.      USE A3U
.
CALL FREEC3(A3,A3I) @ Free storage for A3U
END

SUBROUTINE GETC3(D, IDIM)
CHARACTER*4 D
* GET STORAGE FOR A THREE CHAR/ELT ARRAY.
COMMON /STRG$/INUM3, ISV3A(40), ISV3X(40)
DATA INUM3/0/
IN=MCORF$(IDIM*3/4+2)
IDIM=(IA-LOC(D))*4/3 + 1 @ ELEMENT SEPARATION
IX = 3*IDIM/4 + LOC(D) @ APPROX ADDR
IF (INUM3.GE.40) STOP 'GETC3 MAX'
INUM3=INUM3+1
ISV3A(INUM3)=IA
ISV3X(INUM3)=IX @ APPROX ADDR
END

SUBROUTINE FREEC3(D, IDIM)
CHARACTER*4 D
COMMON /STRG$/INUM3, ISV3A(40), ISV3X(40)
IX=3*IDIM/4+ LOC(D) @ APPROX ADDR
DO 10 I=1,40
IF (ISV3X(I).EQ.IX) THEN
CALL LCORF$(ISV3A(I))
IDIM = 0
INUM3=INUM3-1
IF (I.EQ.INUM3+1.OR.INUM3.EQ.0)RETURN
ISV3A(I)=ISV3A(INUM3+1)
ISV3X(I)=ISV3X(INUM3+1)
RETURN
ENDIF
CONTINUE
STOP 'FREEC3 NO-FIND'
END
```

7.3.3.23. F2DYN\$

Purpose:

This service routine allows the user to have a measure of dynamic storage allocation for dummy arrays.

Form:
$$\text{CALL } \left\{ \begin{array}{l} \text{F2DYN\$} \\ \text{FFDYN\$} \end{array} \right\} (sub, isize_1, \dots, isize_{n-1})$$
where:

sub subprogram name

isize_i integer expression denoting array size in words ($1 \leq i \leq n-1$)

Description:

In the ASCII FORTRAN I/O complex, the storage management routines may be accessed via routine F2DYN\$. It has n arguments. The first argument is the name of an ASCII FORTRAN subprogram to call. The second through n th arguments are the sizes (in words) of $n-1$ arrays which are to be passed to this subprogram. The service subroutine will acquire core via the I/O complex, create $n-1$ dynamic arrays, and pass them on to the subprogram. The subprogram can dimension them dynamically on entry. The acquired storage is released upon the return. The F2DYN\$ subroutine can be called recursively.

In a program system with several subprograms (such as in the following example), the D-bank savings could be substantial because there is much less dead static storage, as a result of a smaller amount of dynamic storage being allocated at any one time.

An alternate entry point FFDYN\$ is provided, so the allocation mechanism can be used for calling both subroutines and functions from the same program unit. Note that the compiler will issue warnings (which can be ignored for this case) if either F2DYN\$ or FFDYN\$ is called from the same program unit with a differing number of arguments.

Example:

Subprogram REDUCE needs some local temporary arrays to do its computations:

```
SUBROUTINE REDUCE(M,MD)
REAL M(MD) @ MD RANGE IS 10 to 1000
DIMENSION M2(1000),M3(1000),M4(1000)
REAL M2
DOUBLE PRECISION M3
COMPLEX*16 M4
.
.
.
END
```

It can be changed to:

```
SUBROUTINE REDUCE(M,MD)
REAL M(MD) @ MD RANGE IS 10 to 1000
COMMON /SIZES/I
EXTERNAL REDINT
I=MD
CALL F2DYN$(REDINT,MD,MD*2,MD*4)
C THE INTERNAL SUB DOES THE PROCESSING NOW.
SUBROUTINE REDINT(M2,M3,M4)
DIMENSION M2(I),M3(I),M4(I)
REAL M2
DOUBLE PRECISION M3
COMPLEX*16 M4
.
.
.
END
```

7.4. Programmer-Defined Procedures

Because of the wide variety of FORTRAN applications, the programmer may desire to use procedures which are not supplied by FORTRAN. Such procedures may be defined using one of the following:

- Statement function
- Function subprogram
- Subroutine subprogram

7.4.1. Statement Functions

A statement function definition specifies an expression to be evaluated whenever that statement function name appears as a function reference in another statement in the same program unit. The expression may involve any appropriate combination of arithmetic, character, and logical operators.

The statement function generates inline code when it is referenced. This allows efficient references to the defining expression without writing the expression each time.

Statement functions are defined using the statement function definition statement.

7.4.1.1. Statement Function Definition Statement

Purpose:

A statement function definition statement associates a statement function name with an expression.

Form:

$$n ([a [, a] \dots]) = exp$$

or:

```
DEFINE n [ ( [ a [ , a ] \dots ] ) ] = exp
```

where:

n is the name of the statement function.

each *a* is a dummy argument used in *exp*. (The maximum number of dummy arguments allowed is 150.)

exp is any FORTRAN expression which references the statement function dummy arguments *a*.

Description:

All statement function definition statements must appear before any executable statements in the program unit, and they should appear after all other specification statements.

The names of each *n* and *a* take the forms of variable names (see 2.2.2.3). An argument, *a*, is a dummy name and is totally independent of any uses of the same name elsewhere in the program unit, except for typing statements. The statement function dummy argument list serves only to indicate order, number, and type of arguments for the statement function.

A type is associated with the name of the statement function and each of its arguments. The type of each name is determined by the normal typing conventions (IMPLICIT and typing statements, and the I-N integer rule). If the type of a statement function is character, its length may be specified as a positive integer constant or as * (see 6.3.2).

Each argument of the statement function should be referred to in *exp*; *exp* may also contain references to other statement functions. However, statement functions may not be referred to before they are defined, and a statement function definition may not refer to itself. In addition, a statement function definition in a function subprogram must not contain a function reference to the name of the function subprogram or an entry name in the function subprogram.

For a description of local-global rules for names used in statement function definition statements in internal subprograms, see 7.11.

Examples:

```
BRIEF(VAR1,VAR2) = (VAR1 + VAR2)/4.
C           Defines a REAL statement function named BRIEF which
C           calculates the sum of its two arguments and divides
C           their sum by four.

INTEGER FLD
FLD(I,J,A) = BITS(A,I+1,J)
C           Defines an integer statement function named FLD whose
C           arguments I, J, and A are used in its expression. The
C           expression is a reference to the BITS pseudo-function.

STAFUN( ) = 2*3-A+482**B-200
C           Defines a REAL statement function with no dummy
C           arguments.

CHARACTER A*4,B*2
DEFINE B(I)=A(I)(3:4)
C           Defines a CHARACTER*2 statement function which takes
C           character positions 3 through 4 of character array
C           element A(I).
```

7.4.1.2. Statement Function References

The reference to a statement function takes the form:

$$n [([e_1 [, e_2] \dots])]$$

where:

n is the name of the statement function.

e_i is an actual argument of the statement function. It must be an expression or an array name.

The order, number, and type of the actual arguments in the statement function reference must be the same as the order, number, and type of the dummy arguments in the statement function definition. If the type of an actual argument does not match the type of the corresponding dummy argument, no conversion will be performed—instead, the actual argument's type will be used in the expression. Note, however, that the type of an actual argument e_i should be consistent with the usage of the corresponding dummy argument a_i in the expression exp in the statement function definition. For example, if .NOT. a_i appears in exp , the corresponding actual argument e_i must be a logical expression.

In effect, each reference to a statement function is replaced by a copy of expression exp in which each occurrence of a dummy argument a is replaced by the corresponding actual argument e . The type of the resulting expression is determined from the types of the actual arguments and the nonargument components of exp according to the rules for expression evaluation (see 2.2.3). When a statement function is referred to, it must generate a legal FORTRAN expression.

If the expression *exp* and the statement function *n* are both arithmetic type (integer, real, or complex), *exp* will be converted, if necessary, to the type of *n*, according to the assignment rules in Table 3-1. If both *exp* and *n* are type character, the length of *exp* will be changed, if necessary, to the length of *n* (as described in Table 3-2). There will be no character length conversion if the statement function is declared as CHARACTER*(*) (see 6.3.2.2).

Note that when an actual argument is an expression, it is evaluated only once and its value is substituted for the dummy argument.

Following its declaration, a statement function may be referenced wherever a reference to a function is permitted.

A statement function reference may sometimes be used as a receiving variable (for example, as the left-hand part of an assignment statement). The latter usage requires that after argument substitution, *exp* must be:

- A simple variable,
- An array element,
- A character substring, or
- The BITS or SUBSTR pseudo-function. (See 7.3.2.1 and 7.3.2.2 for further restrictions on BITS and SUBSTR.)

Note that if a statement function reference appears as a receiving variable, no type conversion is performed between the type of the expression *exp* and the type of statement function *n*.

Example 1:

```
      I(A,J,K,L) = A+J/K**L-A
C          Defines integer statement function I with dummy
C          arguments of type real (A) and integer (J,K,L).
      NEWVAL = I(14.3,-5,2,4)
C          The statement function reference I is evaluated
C          as 14.3+(-5)/2**4-14.3. This expression has a
C          result which is type real. This result is then
C          converted to type integer (since I is type
C          integer). The result of that conversion is stored
C          into NEWVAL (which is type integer).
```

Example 2:

```

C          Suppose that a program requires three 1000 x 1000 logical
C          arrays. Using the FORTRAN LOGICAL data type would
C          require three arrays of 1,000,000 words each. Since
C          ASCII FORTRAN cannot support a single array of more than
C          262,000 words, the program could not be compiled.
C          However, by accepting a slight decrease in readability,
C          the problem can be solved using statement functions.
      INTEGER BIT, L1, L2, L3
      INTEGER LOG1(27778), LOG2(27778), LOG3(27778)
      DEFINE BIT(A,I) = BITS(A((I+35)/36),
1          MOD(I+35,36)+1,1)
```

```
DEFINE L1(I,J) = BIT(LOG1,(I-1)*1000+J)
DEFINE L2(I,J) = BIT(LOG2,(I-1)*1000+J)
DEFINE L3(I,J) = BIT(LOG3,(I-1)*1000+J)
C      Now by arbitrarily associating 0 with the value FALSE
C      and 1 with the value TRUE, L may be used in most of the
C      same ways that a 1000 x 1000 logical array could be used.
C      Thus, the following loop assigns the logical matrix
C      product of L2*L3 to L1.
DO 100 I = 1, 1000, 1
  DO 100 J = 1, 1000, 1
    L1(I,J) = 0
    DO 100 K = 1, 1000, 1
100    L1(I,J) = OR(L1(I,J),AND(L2(I,K),L3(K,J)))
C      Note that while this example is correct, it is not
C      necessarily the most efficient way of performing the
C      desired computation.
```

7.4.2. Function Subprograms

A function subprogram is a separate program unit. It begins with a FUNCTION statement and ends with the next END, SUBROUTINE, or FUNCTION statement. There are external and internal functions. A function is external if it appears as the first program unit or follows an END statement. Otherwise, it is an internal function and is considered as local to the previous external program unit, whether it is a main program, a function, or a subroutine. Note that the FORTRAN 77 standard does not have the concept of an internal subprogram; it is an ASCII FORTRAN extension.

Example:

C Main program

. (block 1)

C Internal function A
FUNCTION A

. (block 2)

C Internal subroutine B
SUBROUTINE B

. (block 3)

END

C External function C
FUNCTION C

. (block 4)

C Internal subroutine D
SUBROUTINE D

. (block 5)

END

```
C External subroutine E
  SUBROUTINE E
```

```
    (block 6)
```

```
  END
```

This compilation unit contains a main program with two internal subprograms (A and B), an external function (C) with one internal subprogram (D), and an external subroutine (E) with no internal subprograms. Each block in the program is a group of statements containing no END, SUBROUTINE, FUNCTION, PROGRAM, or BLOCK DATA statements. See 7.4.3 for a description of subroutines.

7.4.2.1. Structure

The function subprogram may contain any FORTRAN statement other than a BLOCK DATA, SUBROUTINE, or PROGRAM statement. Another FUNCTION or SUBROUTINE statement encountered (with no intervening END statement) causes the following source to be a separate internal subprogram. If an IMPLICIT statement is used in a function subprogram, it should immediately follow the FUNCTION statement.

A function subprogram may not call itself, either directly or indirectly.

For a description of local-global rules for symbolic names used in internal function subprograms, see 7.11.

7.4.2.2. FUNCTION Statement

Purpose:

The FUNCTION statement informs the compiler that the definition of a programmer-written function is being specified.

Form:

```
[ type ] FUNCTION n ( [ a [ , a ] ... ] )
```

or:

```
[ type ] FUNCTION n [*s] [ ( [ a [ , a ] ... ] ) ]
```

where:

type is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, or LOGICAL. This optional field specifies the type of value returned by the function. CHARACTER*s is also allowed, in which case *s is not allowed after n.

n is the symbolic name by which the function is known to other program units. In general, it is not advisable to use the currency symbol (\$) in function names. This avoids conflicts with entry names in the ASCII FORTRAN library and the Series 1100 Operating System relocatable library.

s is one of the length specifications allowed for *type* (see 6.3). This field may only be specified if *type* is specified. If *type* is CHARACTER, *s* may be one of the following: (1) An unsigned, nonzero, integer constant, (2) an integer constant

expression which does not include a parameter constant, enclosed in parentheses, and with a positive value, or (3) an asterisk in parentheses.

each *a* is a dummy argument, which may be a variable name, array name, dummy procedure name, * or \$. If there are no dummy arguments, the parentheses are not required. Any dummy argument may be enclosed in slashes (that is, /*a*/). The number of arguments may not exceed 150. The maximum number of character arguments allowed is 63.

Description:

The FUNCTION statement must be the initial statement of a function subprogram (except possibly for a COMPILER or EDIT statement). It identifies the name of a function, its arguments, and possibly its type and length.

The type of the function (which is the type of the value returned by the function) is determined by the *type* field of the function, an explicit type specification statement within the function subprogram, or using the implicit typing convention described in 2.2.2.2.1. The function must be assigned the same type in all program units which refer to it. If a reference to the function is made from a program unit in which it has a type other than that assigned to the function in the function subprogram, the returned value will not be converted to the type expected by the calling routine. The results of such a call are unpredictable.

The name of the function and each of its ENTRY names (see 7.7) are available throughout the subprogram for use as a variable of the same type as that of the function (or ENTRY name). The function and entry names are effectively equivalenced (see 6.4). If a function is type CHARACTER, then all of its ENTRY names must also be type character.

The function name or an ENTRY name must be assigned a value by appearing on the left-hand side of an assignment statement, as an element of a READ statement list, or as an argument to a subroutine or function which assigns a value to the corresponding formal parameter. The value of the function and ENTRY name may be subsequently referenced or changed. When execution reaches a RETURN or END statement in the FUNCTION subprogram, the value returned is that most recently assigned to any of the associated function or ENTRY names. If the type of the name to which the value was assigned differs from the type of the entry point by which the function is called, no conversion is done. The returned value is then undefined.

The function subprogram may also use one or more of its formal parameters to return values to the calling program unit. See 7.5 for a complete discussion of actual and formal parameters.

For a description of alternate return specifiers (that is, dummy arguments as asterisks), see 7.6. ASCII FORTRAN allows alternate returns from functions, but the FORTRAN 77 standard does not.

7.4.3. Subroutines

A subroutine is a separate program unit. It begins with a SUBROUTINE statement and ends with the next END, SUBROUTINE, or FUNCTION statement. There are internal and external subroutines in exactly the same manner as internal and external functions (see 7.4.2). A subroutine is external if it appears as the first program unit in the source input, or if its SUBROUTINE statement is immediately preceded by an END statement. Otherwise, it is internal.

For a description of local-global rules for symbolic names used in internal subroutine subprograms, see 7.11.

7.4.3.1. Structure

If the subroutine contains an IMPLICIT statement, the IMPLICIT statement should immediately follow the SUBROUTINE statement. The subroutine may contain any FORTRAN statements except a BLOCK DATA, FUNCTION, or PROGRAM statement. Another FUNCTION or SUBROUTINE statement encountered (with no intervening END statement) terminates the subroutine and causes the following source to be an internal program unit.

A subroutine may not invoke itself, either directly or indirectly.

7.4.3.2. SUBROUTINE Statement

Purpose:

The SUBROUTINE statement notifies the compiler that a subroutine is being defined.

Form:

```
SUBROUTINE n [ ([ a [ , a ] ... ] ) ]
```

where:

n is the symbolic name by which the subroutine is known to other program units. In general, it is not advisable to use the currency symbol (\$) in subroutine names. This avoids conflicts with entry names in the ASCII FORTRAN library and the Series 1100 Operating System relocatable library.

each *a* is a dummy argument, which may be a variable name, array name, dummy procedure name, * or \$. A name *a* may only appear once in the list of arguments. The number of arguments may not exceed 150. The maximum number of character arguments allowed is 63. If there are no dummy arguments, the parentheses may be omitted.

Description:

The SUBROUTINE statement must be the first statement in a subroutine subprogram (except possibly for a COMPILER (see 8.5) or EDIT statement (see 8.4)). It specifies the name and arguments of the subroutine identified.

A dummy argument of a subroutine may be a symbolic name, an asterisk, or a currency symbol. Use of an asterisk or a currency symbol indicates that the actual argument corresponding to this dummy argument is a statement label. Such labels may be referenced by the RETURN *i* statement (see 7.6). Dummy arguments which are symbolic names may be enclosed in slashes (/ *a* /).

The subroutine name may not appear in any other statement of the subroutine subprogram.

Example:

```
SUBROUTINE SAMP(CONS, ARR, *)  
C      Defines subroutine SAMP with three arguments. The  
C      asterisk indicates the actual argument is a statement label.
```


7.5. Function and Subroutine Arguments

On each call to a subprogram, the actual arguments are matched to the formal arguments based on their order in the respective list.

The actual arguments must match the formal arguments in number, type, and usage. ASCII FORTRAN will automatically check the types and number of actual arguments against those expected on subprogram entry, except for certain exceptions listed in 8.5.6 (**ARGCHK options of the COMPILER statement**). If a dummy argument is an array, the corresponding actual argument must be either an array or an array element. In the first case, the size of the dummy array should not exceed the size of the actual array. In the second case, the effect is as if the first element of the dummy array were equivalenced to the indicated element of the actual array. The size of the dummy argument should not exceed the size of that portion of the actual array which follows and includes the indicated element.

A dummy argument is an array if it appears in a DIMENSION statement, or with dimensions in an explicit type statement in the subprogram. None of the dummy arguments may appear in an EQUIVALENCE, COMMON, DATA, PARAMETER, SAVE, or INTRINSIC statement.

If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a variable, a character substring, an array element, or an array. A constant or an expression should never be used as an actual argument if the corresponding formal argument is assigned a value. Such an error is not detected by the ASCII FORTRAN system and the results are unpredictable. If an actual argument is in COMMON, or appears more than once in the argument list, undesirable side effects may occur. Consider the following example:

```
SUBROUTINE MULT(A,B,C,LI,LJ,LK)
  DIMENSION A(LI,LJ),B(LJ,LK),C(LI,LK)
  DO 1 I = 1, LI
    DO 1 K = 1, LK
      C(I,K) = 0.0
    DO 1 J = 1, LJ
      C(I,K) = A(I,J)*B(J,K)+C(I,K)
1  RETURN
  END
```

If this subroutine is called by:

```
CALL MULT(X,Y,X,5,5,5)
```

then the result of the call will be incorrect since X is modified before the multiplication is complete. Another type of problem is illustrated in the following:

```
SUBROUTINE FC3(A,B,C)
  A = B + C
  C = SQRT(A**2+B**2+C**2)
  A = A - B
  END
```

If the call is:

```
CALL FC3(X,Y,X)
```

then the result will depend on whether or not optimization is requested. If there is no optimization, the code executed is:

```
X = Y + X  
X = SQRT(X**2 + Y**2 + X**2)  
X = X - Y
```

If optimization is requested, the code executed is:

```
T1 = Y + X  
X = SQRT(T1*T1 + Y*Y + X*X)  
X = T1 - Y
```

Similar results can happen if the dummy argument to which a value is assigned is associated via a call with a variable in a COMMON block.

The compiler generates code for a subprogram, assuming that its arguments change their values only through explicit statements in the subprogram. If an argument is actually in a common block, and another subprogram is called which changes its value, the execution of the first subprogram may or may not reflect this change in the argument value. **The same situation exists on execution of a Direct Access I/O statement when the associated variable is unknown for the statement either because of a variable being used for a unit number, or because no OPEN or DEFINE FILE statement on the unit number exists in the subprogram. The compiler may assume that the associated variable does not change its value, since it does not realize it is the associated variable.**

7.6. RETURN Statement

Purpose:

The RETURN statement provides a mechanism for returning control to the calling routine from a point in the subprogram other than the END statement.

Form:

```
RETURN [ / ]
```

where i is an optional integer expression whose value, for example n , denotes the n^{th} statement label in the argument list.

Description:

A RETURN statement in a main program is equivalent to a STOP statement (see 4.8).

When the RETURN statement is executed in a function subprogram, evaluation of the referencing expression is resumed with the function name replaced by its returned value. However, if i is specified, execution continues at the statement whose label is selected by i . This results in a loss of the value normally returned from a function reference.

Execution of a RETURN statement in a subroutine subprogram ordinarily causes execution to continue following the CALL statement. However, if i is specified, execution continues at the statement whose label is selected by i . The value of i must be positive and no greater than the number of statement labels passed as arguments.

If the value of i is outside of the correct range, a warning message will be printed at run time:

```
RETURN ARGUMENT / OUT OF RANGE
```

A return will then be executed as if i had not been specified.

Examples:

```
SUBROUTINE SCALAR(I, K, A, B, IMAX, KMAX, N, TEST, *, *)
```

```
  .  
  .  
  .  
RETURN 1
```

```
  .  
  .  
RETURN
```

C The first return passes control to the statement identified
C by the first label in the argument list, which is the
C ninth argument of subroutine SCALAR (see 7.4.3.2). The
C second return passes program control to the point
C where the subroutine was invoked.

7.7. ENTRY Statement

Purpose:

The ENTRY statement defines an alternate point to begin execution of a subroutine or function subprogram, and an alternate name by which the subprogram may be referenced. The ENTRY statement can also be used to allow a function subprogram to return values of different types.

Form:

```
ENTRY n [ ( [ a [ , a ] ... ) ] ]
```

Description:

The entry name *n* is a symbolic name which may be referenced subject to the same rules as the corresponding subroutine or function name. As with subroutine and function names, the currency symbol (\$) should be avoided in entry names; it is nonstandard.

Each formal argument *a* is a symbolic name, which must be unique in the formal argument list. The formal arguments need not agree with those of the SUBROUTINE or FUNCTION statement, or any other ENTRY statement, in order, number, type, or usage. A formal argument which appears in more than one formal argument list need not occupy the same position in each list. The use of slashes (/a/) is permitted as it is for a SUBROUTINE or FUNCTION statement. A maximum of 150 formal argument names may appear in a program unit (SUBROUTINE, FUNCTION, and all ENTRY statements). The maximum number of type character formal arguments allowed in a single ENTRY statement is 63.

Any formal argument may be an asterisk or a currency symbol to denote that the corresponding actual argument is a statement label. (The FORTRAN 77 standard limits the use of asterisks for formal arguments to subroutines, though ASCII FORTRAN allows them in functions also.)

All references to a formal argument name, or an entry name, in executable statements must follow the first appearance of the name in a formal argument list, or the ENTRY statement which defines the entry name.

A program unit may contain multiple ENTRY statements. Each defines an alternate entry point to the subroutine or function and defines the name by which the entry point can be referenced. An entry point in a subroutine must be referenced as a subroutine; an entry point in a function must be referenced as a function. A total of no more than 511 entry points may be specified for all program units in a compilation.

ENTRY statements are not themselves executable. If the normal sequence of statement execution would cause an ENTRY statement to be executed, it would be skipped and the next executable statement would be executed. When an ENTRY-defined name is referenced (in a CALL statement or function reference), execution of the subprogram begins with the first executable statement following the ENTRY statement. An ENTRY statement must not appear in the range of a DO-loop or in a block IF structure (that is, between a block IF statement and its corresponding END IF statement, where the IF-level is greater than zero).

If the ENTRY statement appears in a function subprogram, the entry name is available for use as a variable in the same manner as the function name; the entry names and function name are effectively equivalenced. The type of each function entry name is determined from explicit typing, the IMPLICIT statement, or the I-N integer rule, whichever applies to the name. At the time of return, if the last function/entry name variable assigned is of a different type than that required for the entry name referenced, the function value is undefined. If a function is type character, then all of its entry names must be type character, but all of its entry names are not required to be of equal length. A reference

to a character function name or character entry name always uses the character length associated with the name most recently entered.

FORTRAN 77 requires that if a function is of type character with a length of *, all ENTRY names must also be type character and of length *; otherwise, the character length must be the same integer value for each ENTRY name.

A subprogram may not reference itself, by the subprogram name or any of its entry names, either directly or indirectly. Doing so cannot always be detected by the compiler and will result in an infinite loop at execution time.

If information for the object-time dimensioning of an array is passed in a reference to an ENTRY statement, the array name and all of its dimension parameters (except any that are in a common block) must appear in the argument list of the ENTRY statement.

When dummy arguments are not in the dummy argument list of the name through which the subprogram was referred to, they generally refer to the actual arguments with which they were most recently matched. The compiler cannot detect this situation; do not attempt to use it. Problems which can occur include:

- If an actual argument is an expression, its value is passed in a temporary area. Since this area is reused for following statements, its value may change even though the expression would have the same value if it were reevaluated.
- If the segmentation facility of the Collector is in use, the storage occupied by the actual argument may contain part of a different program unit when the subprogram is next called.
- If automatic storage is used, the dummy argument may find different contents on the stack the next time the subprogram is called.
- If the actual argument is a variable, the dummy argument remains associated with that variable, and reflects any changes to the value of that variable. As a result, the dummy argument may not retain the value that it had at the last execution of the subprogram.
- A dummy argument may not be used unless an entry point has previously been entered where it exists in the argument list.
- Optimization generally cannot know that this side effect is taking place, and will produce code as if it had not taken place.

7.8. BLOCK DATA Subprograms

A BLOCK DATA procedure allows the initializations of variables in blank or labeled common blocks to be combined in a single program unit. ASCII FORTRAN allows the initialization of blank common, but the FORTRAN 77 standard does not.

7.8.1. Structure

A BLOCK DATA procedure is a separate program unit. It must begin with a BLOCK DATA statement (see 7.8.2). The only other statements which are permitted in a BLOCK DATA subprogram are IMPLICIT, PARAMETER, DIMENSION, COMMON, SAVE, EQUIVALENCE, DATA, END, typing statements, and the EXTERNAL and INTRINSIC statements.

Since the association between COMMON variables of different program units is by position in a COMMON block, rather than by name, all elements of a COMMON block should be listed in the COMMON statement, even though they are not all initialized.

FORTRAN 77 allows named common blocks to be initialized only in BLOCK DATA subprograms. ASCII FORTRAN allows any program unit (including BLOCK DATA procedures) to initialize elements in one or more common blocks, and a common block may be initialized by more than one program unit. If an element of a common block is initialized by more than one program unit, or more than once within a program unit, one of the initial values will be selected when the program is collected. Unless the initial values assigned to the element are all identical, the actual value assigned to the element at the beginning of execution is unpredictable.

A BLOCK DATA procedure requires no local storage. Any variable which appears in specification statements, but not in a COMMON statement, will be noted in a warning message. No storage will be reserved for variables which do not appear in a COMMON statement.

Since a BLOCK DATA procedure defines no entry points, the automatic element inclusion facilities of the Collector will never cause inclusion of the relocatable generated for a BLOCK DATA procedure. As a result, the use of a BLOCK DATA procedure requires that the collection include an IN directive for the corresponding relocatable element.

7.8.2. BLOCK DATA Statement

Purpose:

A BLOCK DATA statement identifies a program unit as a BLOCK DATA subprogram.

Form:

```
BLOCK DATA [ sub ]
```

where *sub* is an optional name for the BLOCK DATA subprogram.

Description:

The BLOCK DATA statement is the initial statement in all BLOCK DATA subprograms.

The name *sub* is a global name and must not be the same as the name of an external procedure, main program, common block, or other BLOCK DATA subprogram in the same executable program. In addition, *sub* must not be the same as any local name in the subprogram.

There must not be more than one unnamed BLOCK DATA subprogram in an executable program.

The name of a BLOCK DATA subprogram has no explicit use within the FORTRAN language. It is available mainly for documentation.

The same common block should not be specified in more than one BLOCK DATA subprogram in the same executable program.

Examples:

```
BLOCK DATA BLKA
DIMENSION K(10)
COMMON /B/K
DATA K/1,2,3,4,5,6,7,8,9,10/
END
```

C This exemplifies a BLOCK DATA subprogram named BLKA.
C The COMMON block named B is initialized by the
C DATA statement.

```
BLOCK DATA
DIMENSION A(10),M(10)
COMMON /X/ A,B,C /NAME1/M
DATA M/2*1, 2*3, 2*5, 2*2, 2*4/,B,C/1.0,2.0/
END
```

C The elements of M are initialized to the sequence of
C values (1,1,3,3,5,5,2,2,4,4). The variables B and C
C are set to 1.0 and 2.0, respectively. No initial value
C is assigned to any element of A.

7.9. PROGRAM Statement

Purpose:

A PROGRAM statement is optional and is used to associate a name with the main program.

Form:

```
PROGRAM pgm
```

where *pgm* is the symbolic name of the main program in which the PROGRAM statement appears.

Description:

A PROGRAM statement is not required to appear in an executable program. However, if it does appear, it must be the first statement of the main program.

The symbolic name *pgm* is global to the executable program. It must be unique relative to external procedure names, BLOCK DATA subprogram names, and common block names in the same executable program. In addition, *pgm* must not duplicate any local name in the main program.

The name of a main program has no explicit use within the FORTRAN language. It is available mainly for documentation.

7.10. Non-FORTRAN Procedures

Because FORTRAN is a high-level language oriented to the solution of certain types of problems, it is sometimes expedient to code some parts of a program in FORTRAN and other parts in another language.

These non-FORTRAN parts can be included in the FORTRAN program unit when the program is collected. Program parts written in assembly language (that is, MASM) must be prepared to accept the calling sequences generated by the ASCII FORTRAN compiler. Parts written in SPERRY UNIVAC FORTRAN V, ASCII COBOL, or PL/I may be called directly (see EXTERNAL statement (7.2.3)).

See Appendix K, Interlanguage Communication, for details on the interfaces with FORTRAN V, ASCII COBOL, PL/I, and MASM.

7.11. Scope of Names (Local - Global Definitions)

Entities that may be referred to from both external and internal program units are referred to as "global". Entities that may be referred to only within a particular internal subprogram are referred to as "local", that is, local to that subprogram.

Entities used in an external program unit cannot be referred to by another external program unit unless they are in COMMON blocks or passed as arguments. Variables, arrays, statement functions, parameter constants, and so forth, which are declared or used in an external program unit can be referred to by any of its internal subprograms. Internal subprograms cannot reference each other's data except through COMMON blocks and arguments.

External program units may contain any number of internal subprograms. An internal subprogram may call another internal subprogram if both are within the same external program unit. BLOCK DATA subprograms cannot be internal and cannot contain internal subprograms.

Any data declarations encountered in an internal subprogram create variables local to that internal subprogram. The following statements create local names when in an internal subprogram:

- Explicit typing statements (such as REAL, INTEGER)
- DIMENSION (the array names)
- COMMON (the variable names)
- BANK (the bank names)
- NAMELIST (the namelist name)
- PARAMETER (the name being defined)
- EXTERNAL
- DEFINE (the statement function name)
- Formal arguments in FUNCTION, SUBROUTINE, and ENTRY statements

For rules on IMPLICIT type associations in internal subprograms, see 6.3.1.

Note that a COMMON statement in an internal subprogram redefines (that is, is not a continuation of) the same common block declared in the external program unit.

In addition, all statements create local variables for names encountered which do not exist in their external program unit.

Note that names that are used in an internal subprogram, but are not defined locally in the aforementioned statements, will come from the external program unit environment if they exist there. If they are to be local to the internal subprogram, their declarations must occur before they are referred to.

Example 1:

```
INTEGER P
DIMENSION X(5)
PARAMETER (P=10)
CALL INT(X)
SUBROUTINE INT(Y)
DIMENSION Y(P)           @ uses 10
PARAMETER (P=5)         @ does not help Y
.
.
.
```

Example 2:

```
A=10.                   @ global A
CALL INT
SUBROUTINE INT
NAMELIST/LIST/A,B      @ global A, local B
REAL A                 @ local A
B,A=3.                 @ local A
WRITE(6,LIST)          @ write global A, local B
END
```

Statement labels are local to the program unit in which they appear.

An internal subprogram cannot be referred to by any external program unit outside its program unit group (a program unit group is composed of the external program unit and any internal subprograms contained in the external program unit).

However, if the name of an internal subprogram is passed as an argument to an external program unit outside its program unit group, the external program unit can, in general, refer to the internal subprogram by referring to the dummy argument name. There are two exceptions to referring to an internal subprogram through an argument name:

- In a program compiled with the automatic storage stack feature (COMPILER statement options DATA=AUTO or DATA=REUSE), an external program unit can refer to internal subprograms only from its program unit group. This restriction is required because stack storage is acquired for all parts of a program unit group at one time upon entry to the external program unit.
- In multibanked programs, an internal subprogram cannot be called from a program unit that is not in the same I-bank as the internal subprogram. This restriction is required because internal subprograms do not use an LIJ for returns; they always jump back.

If an intrinsic function name such as SIN is to be used as an internal function, it must be placed in an EXTERNAL statement in its external program unit. This is a special use of the EXTERNAL statement to indicate that the intrinsic function is not desired. In this case, all references to the function SIN in the external program unit or in any of its internal subprograms would refer to the internal subprogram SIN.

7.12. SAVE Statement

Purpose:

A SAVE statement is used to retain the values of entities as they were defined after the execution of a RETURN or END statement in a subprogram. Upon reentry of the subprogram, an entity specified by a SAVE statement will contain the value as last defined by the execution of the subprogram.

Form:

```
SAVE [ n [ , n ] ... ]
```

where n is a named common block preceded and followed by a slash, a variable name, or an array name.

Description:

A SAVE statement is nonexecutable. Within a function or subroutine, an entity specified in a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram. However, an entity in a common block or a global entity in an internal subprogram may become redefined in another program unit.

Dummy argument 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 allowable items in that program unit. A SAVE statement is optional in a main program and has no effect.

The appearance of a common block name preceded and followed by a slash in a SAVE statement has the effect of specifying all of the entities in that common block.

The SAVE statement operates by ensuring static storage for the named or implied entities. ASCII FORTRAN normally has static storage for common blocks. However, if the COMPILER statement with DATA=AUTO is present, local variables are not saved upon execution of a RETURN or END statement, since storage is dynamically acquired. The SAVE statement will cause local variables to be placed under location counter 8, which is static storage in a program that contains the DATA=AUTO option. If the Collector segmentation facilities are used to collect a program, the SAVE statement does not ensure that these entities will be preserved, since ASCII FORTRAN has no control over the collection. Overlays of storage by any means, including Collector segmentation, may cause the loss of values of the entities named in SAVE statements in the overlaid programs.

8. Program Control Statements

8.1. General

The program control statements are used for the temporary modification of the source program being compiled, for modifying the code generated, and for controlling the compilation listing produced. The statements (all nonstandard) are:

- INCLUDE
- DELETE
- EDIT
- COMPILER

The first two statements are used, respectively, to insert additional statements into the compilation, and to delete source statements from the compilation. The next statement dynamically controls the compilation listing. The fourth statement is used to alter the compilation process.

8.2. INCLUDE Statement

Purpose:

The INCLUDE statement inserts an externally defined set of ASCII FORTRAN statements into the program being compiled.

Form:

```
INCLUDE [f.]n [, LIST]
```

where:

- f* if specified, is a file name specifying the file to be searched for the procedure (PROC)
n
- n* is the name of a FORTRAN procedure (PROC) created by the Procedure Definition Processor (PDP). See the EXEC System Utilities Programmer Reference, UP-4144.3 (see Preface), for details on PDP. The name *n* contains from 1 to 12 characters from the same character set as a FORTRAN symbolic name.
- LIST when entered, will cause the included statements to be listed whenever the source program is listed.

Description:

The included statements do not become part of the updated source program produced. Therefore, the physical line numbers of the source program are not changed.

INCLUDE statements may not be nested; that is, an INCLUDE statement must not be among a group of statements to be included (any other FORTRAN statement may be included).

A normal application of the INCLUDE statement is the representation of a block of FORTRAN source statements by a single INCLUDE statement. For example, the inclusion of a set of statement functions which are frequently used at a particular installation is such an application.

A useful application for INCLUDE procedures is one or more INCLUDE elements containing a set of data declarations shared between different portions of a larger user program system. These data declarations could be, for example, many of the COMMON, DIMENSION, EQUIVALENCE, and PARAMETER statements as well as statement function definitions. Caution is urged, however, to the extent to which these FORTRAN procedures are used. All of the statements must be processed, slowing down the compilation. An example of poor usage is a small subroutine including a procedure (PROC) which defines several hundred variables, but uses only one of these definitions.

The Procedure Definition Processor (PDP) is used to create a FORTRAN procedure (PROC) (see Figure 8-1). This is a set of FORTRAN statements which can be inserted into source language with an INCLUDE statement. The PDP accepts source language statements defining FORTRAN procedures and builds an element in the user-defined program file. Using INCLUDE, these procedures may be referenced subsequently in a compilation without redefinition.

A FORTRAN procedure has the following format:

1. The PROC line is the first line of a FORTRAN procedure. It contains the procedure name (1 to 12 characters), starting in column 1, and "PROC" starting in column 7 or after.
2. The procedure images follow the PROC line. These images are the FORTRAN statements which are to be included.
3. The final line must be the word END appearing in columns 2 through 4.

A PDP source element may contain more than one FORTRAN procedure, as in Figure 8-1.

A FORTRAN procedure is not analogous to an Assembler procedure. For a FORTRAN procedure, the F option must be in the @PDP command.

The sample procedures in Figure 8-1 (SPECS1, SPECS2, and SPECS3) could be included (via the INCLUDE statement) in a FORTRAN program.

```
@PDP,FIL STUFF
PDP12R1 R72-16 02/09/77 09:18:37 (,0) RI

PE0001  SPECS1 PROC
0002      IMPLICIT INTEGER (0-0)
0003      PARAMETER (IN=5, OUT=6)
0004      PARAMETER (P1=1,P2=2,P3=3)
0005      PARAMETER (M=MAX(MOD(P3,P2),DIM(P3,P1)))
0006      COMPLEX C(10),D(5)/5*(1,0,-1.0)/
0007      DIMENSION O(5),P(4)
0008      COMMON /CB1/C,K,C / /A(5),Q
0009
0010  9999  FORMAT( )
0011      EXTERNAL FUNEX
0012      END
PE0013  SPECS2 PROC
0014      COMPLEX X(5)
0015      COMMON E,F,G
0016      END
PE0017  SPECS3 PROC
0018      COMPLEX C
0019      COMMON /CB1/ C(5),K,Q
0020      END
```

Figure 8-1. Sample PROC

Example:

```
INCLUDE SPECS1

INCLUDE SPECS3,LIST
```

In order for PDP to add the procedure names to the FORTRAN Procedure Table of the specified program file, one of three conditions must be satisfied:

- The I option must be specified and an element name given in the *spec1* field of the PDP control card; or
- The U option must be specified and an element name given in the *spec1* field of the PDP control card; or
- Neither the I nor the U option is specified and element names are given in the *spec1* and *spec2* fields of the PDP control card.

If a file name is specified in the INCLUDE statement, the FORTRAN Procedure Table of that file is searched for the PROC *n*. The file name is not checked for syntax before it is passed to the Executive, so errors in specifying the file name may result in immediate termination of the compilation by the Executive. The file name may be a fully specified name (qualifier, file, cycle, keys) or a @USE attached name. The usual Executive file name dropout rules apply.

If no file name is given in the INCLUDE statement, the PDP element that contains the procedure (PROC) to be included must be in a file that is assigned to the run in which the compilation is being performed. The files that are searched and the rules for determining the order in which the search is performed are defined as follows:

- If the source input is coming from a program file on mass storage, that file is searched.
- If the source input is coming from a program file on tape, the relocatable binary output file is searched.
- If the source input is coming from cards, the source output file is searched, if one exists. Otherwise, the relocatable binary output file is searched.
- If the PROC is not found in any of the files previously listed, the system library file (SYS\$*RLIB\$) is searched.
- If the PROC is not found in any of the files previously listed, an error is printed. Compilation continues as if the INCLUDE statement had not been present.

A file may be provided which would be searched first to find a PROC to be included. The optional search is achieved by assigning the mass storage file with the @USE attached name FTN\$PF in the run and by specifying the M option on the @FTN statement for the compilations in which the optional file search is desired.

If the file FTN\$PF is assigned to the run and the M option is specified, FTN\$PF will be the first file searched for the PROC to be included. If FTN\$PF is not assigned to the run, or the M option is not specified, or the PROC is not found in FTN\$PF, the PROC is searched for as previously specified.

8.3. DELETE Statement

Purpose:

The DELETE statement prevents compilation of a set of source lines contained in the input source program.

Form:

```
DELETE n [ , [d] [ / ] ]
```

where:

n is a statement label in the same program unit as the DELETE statement.

d is the delete parameter. It must be an integer constant or integer PARAMETER constant.

l is the list parameter. It must be an integer constant or integer PARAMETER constant.

Description:

The DELETE statement causes the compiler to ignore all source lines following the DELETE statement up to, but not including, the statement labeled *n*. However, if the value of the delete parameter, *d*, is zero, the DELETE statement is ignored.

If the value of the list parameter, *l*, is nonzero, the deleted source lines are listed in the compilation listing. If the value of *l* is zero, the deleted lines are not listed.

The lines deleted are made transparent to the compiler, but they are not actually removed from the updated source program.

Several simplifications of the form of the DELETE statement are allowable when the compiler-specified values of the delete and/or list parameters are desired. The compiler-specified values are *d* = 1 and *l* = 0. If both of these values are desired for a particular DELETE statement, the following reduced form may be used:

```
DELETE n
```

If the deleted lines are never to be listed (*l* = 0), the following form may be used:

```
DELETE n,d
```

If the images are always to be deleted (*d* = 1), and the list option is to be specified, the following form may be used:

```
DELETE n , /l
```

Note that any statement (including a specification statement) may have a label (see 10.3.1), so DELETE *n* may refer to any statement in the program unit.

8.4. EDIT Statement

Purpose:

The EDIT statement controls the type of compilation listing produced for any portion of the source program.

Forms:

```
START EDIT PAGE  
START EDIT SOURCE  
START EDIT CODE  
STOP EDIT SOURCE  
STOP EDIT CODE
```

Description:

The word PAGE causes a page eject in the source listing, the word SOURCE controls the listing of the source program statements, and the word CODE controls the listing of the generated object code.

The START forms cause the compiler to initiate the type of listing specified (SOURCE or CODE), if it has been suppressed by a preceding EDIT statement or by a processor call option. A START EDIT CODE statement initiates both source and code listings.

The STOP forms cause the compiler to terminate the type of listing specified, if it has been initiated by a preceding EDIT statement or by a processor call option. A STOP EDIT SOURCE statement terminates both source and code listings.

The START EDIT PAGE form causes the current page of the compilation listing to be ejected before listing the next statement of the source program. This has the effect of placing the next statement following this command at the top of a new page. If the source is currently not being printed, this statement has no effect on the output.

8.5. COMPILER Statement

Purpose:

The COMPILER statement provides options to the compiler which are intrinsic to the program.

Form:

```
COMPILER (op) [ . (op) ] . . .
```

where *op* is a compilation option.

Description:

A programmer may choose any of the following options for the compilation process of a program unit by using the COMPILER statement. All options are of the form $a = b$, where a is the option name and b is the option type.

The allowable options are:

- DATA = AUTO
- DATA = REUSE
- PARMINIT = INLINE
- BANKED = RETURN
- BANKED = DUMARG
- BANKED = ACTARG
- BANKED = ALL
- LINK = IBJ\$
- STD = 66
- U1110 = OPT
- ARGCHK = ON
- ARGCHK = OFF
- PROGRAM = BIG

If the COMPILER statement is used, it affects the code generated for the entire program unit (or compilation), and it must be the first statement of the unit (or compilation).

The BANKED = RETURN option is local to the program unit in which it appears. All other options are global to the compilation, and should appear at the top of the first program unit in the source.

8.5.1. DATA and PARMINIT Options

The DATA=AUTO and DATA=REUSE options cause the ASCII FORTRAN compiler to generate code which has no local D-bank (other than COMMON blocks), to acquire and release space for local use via calls to run-time routines, and to dynamically initialize this space on entry, thus giving ASCII FORTRAN an automatic storage facility.

These two options also cause the LINK=IBJ\$ option to be turned on. The DATA=REUSE option is identical to the DATA=AUTO option, except that a run-time Prolog entry point is used which does not initially push the stack before allocating local storage. This means that the stack storage of the previous routine is reused. This conserves stack space, but also means that a RETURN can never be done from a subprogram with a DATA=REUSE option, since the stack of its caller is destroyed. The user would have to somehow handle program termination himself. Since arguments are passed in the stack, arguments cannot be passed to routines which have the DATA=REUSE option; that is, they must not have parameters.

The PARMINIT=INLINE option has effect only when used with the DATA=AUTO or DATA=REUSE option. When the DATA=AUTO or DATA=REUSE option is specified, a considerable amount of initialization code is required for the automatic storage stack. Items such as I/O packets, format lists, parameter lists, and data lists require instructions to initialize the list information onto the automatic storage stack. This initialization code is normally executed for the entire program unit upon entry to the main program or subprogram. That is, all of the stack initialization for the entire program unit is executed at program initialization before executing any of the statements of the program. This is inefficient if only a small part of the program is executed each time. The PARMINIT=INLINE option can be used to make certain FORTRAN programs execute more efficiently. This option causes the initialization of parameter lists to take place at their reference, rather than at program initialization time. This is more efficient for programs that execute only a portion of their statements each time called and do not loop heavily around subprogram calls. However, if the program has many loops, with subprogram calls within the loops, the use of this option could result in more execution time instead of less.

Because of the highly volatile nature of this automatic storage, some useful side effects of static storage allocation are not available when using automatic storage:

- Local variables not initialized by DATA statements will not have the value of zero on entry.
- Local variables will not have their last value on reentry to a subprogram, unless they appear in a SAVE statement.
- Arguments set by entering at one entry point are not available when the subprogram is entered at another.
- If one reads into a Hollerith or literal field of a FORMAT statement it will hold that value only as long as the subprogram containing it is active.
- Names of internal subprograms cannot be passed to other external subprograms and have the external call the internal.

Full-debug checkout and FTNPMD (see 10.6 and 10.7) use is severely restricted when using automatic storage:

- Walkback will terminate when a subprogram using automatic storage is encountered.
- No local variables or arguments can be set or dumped for subprograms which use automatic storage.

8.5.2. BANKED Options

The ASCII FORTRAN banking mechanism is described in H.2.

■ BANKED=ALL

BANKED=ALL is a general banking option used to indicate that banking of some form will appear in the user program.

The option indicates that all labeled common blocks may be in paged data banks or in the control bank. In addition, BANKED=ALL turns on the following COMPILER statement banking options:

LINK=IBJ\$ (subprograms may be banked)
BANKED=DUMARG (input arguments may be banked)

If BANKED=ALL appears, the specific subprogram (I-bank) and common block (D-bank) bank structure of the user program need not be known at compile time. This makes the process of generating a multi-bank user program more flexible, since changes in the bank structure can be made entirely at collection time, with no ASCII FORTRAN recompilations required.

If the user has paged data banks and BANKED=ALL is specified, BANK statements (see 6.6) naming common blocks may be completely omitted from ASCII FORTRAN source programs, although they may appear for reasons of efficiency. See H.2.3 for a description of banking efficiency.

■ BANKED=DUMARG

The BANKED=DUMARG option applies to dummy arguments in subprograms. (A dummy argument is an item appearing in argument lists in SUBROUTINE, FUNCTION, or ENTRY statements.) When this option appears, dummy arguments which are data items may be in paged data banks or in the control bank, and dummy arguments which are subprograms may be in any I-bank or in the control bank.

On every reference to a dummy argument in a subprogram, the compiler must generate code to base the item's bank, if it is not already based.

The BANKED=DUMARG feature is automatically turned on if BANKED=ALL is specified.

■ BANKED=RETURN

When this option appears in a subprogram, the subprogram returns to its calling program via one of two instructions: LIJ or a jump. The decision is made in generated code with a test sequence: an LIJ is executed if the subprogram was entered via an LIJ, and a jump is executed if the subprogram was entered via an LMJ.

This feature should be used if the subprogram's I-bank may be different from the I-bank of any calling program.

The BANKED=RETURN feature is automatically turned on if LINK=IBJ\$ or BANKED=ALL is specified.

■ BANKED=ACTARG

BANKED=ACTARG is an obsolete option that is supplied for compatibility with previous levels of ASCII FORTRAN. It previously was used to pass banking information for arguments on subprogram calls, but this information is automatically passed now (if the user specifies any form of banking, for example, BANKED=ALL).

8.5.3. LINK=IBJ\$ Option

The LINK=IBJ\$ option is provided for the purpose of making it easier to construct multi-bank programs without the use of the COMPILER (BANKED=RETURN) or BANK statements. This option is meaningful for banked subprograms, although it may also be used on nonbanked programs. The LINK=IBJ\$ option has no effect on paged data banks.

See H.2 for a description of the ASCII FORTRAN banking mechanism. Note that the LINK=IBJ\$ feature is automatically turned on if BANKED=ALL is specified.

The LINK=IBJ\$ option causes the compiler to generate the following reference to user subprograms:

LXI,U	X11, BDICALL\$+ <i>subprg</i>
IBJ\$	X11, <i>subprg</i>

BDICALL\$ and IBJ\$ are special external reference items which are resolved at collection time; *subprg* represents the name of a FORTRAN external subprogram. The Collector determines if *subprg* is contained in a different instruction bank than the one from which the call is made. If it is, then BDICALL\$+ *subprg* is the BDI of the bank in which *subprg* is located, and IBJ\$ is an LIJ. If *subprg* is in the same instruction bank, BDICALL\$+ *subprg* is zero, and IBJ\$ is an LMJ. The LINK=IBJ\$ option also causes the compiler to generate the following instructions to return from subprograms:

LA,H1	A4, X11-save-location
JZ	A4,0,X11
LXI	X11,A4
LIJ	X11,0,X11

Use of the LINK=IBJ\$ option, as opposed to the BANKED=RETURN option and BANK statement, also allows for modification of bank structure collections without change to the source program, and thus without recompilation. An IBJ\$ reference will do an LMJ or LIJ to the correct bank, and the subprogram will return correctly regardless of the bank structure.

8.5.4. U1110 = OPT Option

The U1110=OPT option invokes a code reordering algorithm during the code generation phase of the compiler. This option is meaningful only on hardware which incurs register conflicts, such as the SPERRY UNIVAC 1100/40, 1100/60, and 1100/80 Systems. It involves shuffling generated code, which minimizes the hardware register conflicts in order to obtain faster execution speeds. It is also available on older machines, such as the 1106 System, so that absolutes and relocatables created on the older machines will execute faster if moved to a newer machine.

Example of code reordering:

Source:

```
J = I
N = K + M
P = O
```

Generated code:

```
1) L  A4,I
2) S  A4,J
3) L  A6,K
4) A  A6,M
5) S  A6,N
6) SZ P
```

There would be a major register conflict between lines 1 and 2, and between lines 4 and 5. A large delay would occur at each of the two points. After code reordering, the sequence would be:

```
1) L  A4,I
2) L  A6,K
3) A  A6,M
4) SZ P
5) S  A4,J
6) S  A6,N
```

Here there would remain a small delay between lines 5 and 6, but this sequence is much faster than the unordered code.

The algorithm used is a simple look-ahead that stops when labels, calls, etc. are encountered. In general, a decrease of 3% to 8% in Central Arithmetic Unit (CAU) time can be expected for execution of the generated code. (However, I/O-bound programs may see no change and a heavily looping program may execute up to 30% faster.) The effect is most pronounced on programs that also are compiled with global optimization (the Z option).

This code-reordering algorithm is also invoked by the use of the E option on the ASCII FORTRAN processor call statement (@FTN,E).

8.5.5. STD=66 Option

The STD=66 option is provided to allow relocatables created with level 8R1 and relocatables created with level 9R1 and higher to coexist.

ASCII FORTRAN levels 9R1 and higher conform to the latest FORTRAN standard, FORTRAN 77 (formally known as ANSI X3.9-1978). ASCII FORTRAN levels lower than 9R1 conformed to ANSI X3.9-1966.

Compatibility problems arise because conforming with FORTRAN 77 requires certain features that conflict with the implementation of previous ASCII FORTRAN levels.

If the option STD=66 is not specified in a compilation, ASCII FORTRAN will conform to the standard (FORTRAN 77). If the option is specified, ASCII FORTRAN will implement the features in Table 8-1 as they were done in previous levels (that is, in a nonstandard manner).

If any pre-level 9R1 ASCII FORTRAN relocatables are used in a collection, the ASCII FORTRAN level 9R1 or 10R1 compilations must be done with the COMPILER (STD=66) compatibility option. Also, all relocatables in a collected program must have matching compatibility options. If one program is compiled with STD=66, all FORTRAN programs to be collected with that program must have STD=66.

Table 8-1. Use of the STD=66 COMPILER Statement Option

Feature	Implementation With STD=66 Option	Implementation Without STD=66 Option
Character data	Unpacked character data - all character items (including array elements) begin on word boundaries.	Packed character data for arrays, common blocks, etc. - character items need not begin on word boundaries (see 6.9.1). If either a dummy or actual argument is type character, then the other must also be type character (see 7.5).
DO-loops	A DO-loop (with DO statement $DO\ n\ i = e_1, e_2, e_3$) must be traversed at least once, even if $e_1 > e_2$ and $e_3 > 0$. -1 is assumed as the increment value if e_3 is not specified, and e_1 and e_2 are both constant expressions, where $e_1 > e_2$.	A DO-loop is traversed zero times if $e_1 > e_2$ and $e_3 > 0$, or if $e_1 < e_2$ and $e_3 < 0$ (see 4.5.4.1). 1 is assumed as the increment value if e_3 is not specified.
Typing	Parameter constants and statement functions have no associated types. Types are determined by usage in expressions.	Parameter constants (see 6.7) and statement functions (see 7.4.1) have associated types, based on normal typing conventions, and previous IMPLICIT and typing statements.

8.5.6. ARGCHK Options

ASCII FORTRAN will automatically check the types and number of actual arguments against those expected on subprogram entry. However, if the Z or V options are specified on the @FTN processor call (optimization options), type checking will not be done. If type checking is desired when optimization is used, the ARGCHK=ON option must be used.

If type checking is not desired, the ARGCHK=OFF option can be used. Note that if either the calling routine or the routine being called has disabled type checking by using the ARGCHK=OFF option or by compiling with optimization (without the ARGCHK=ON option), type checking will not be done.

8.5.7. PROGRAM=BIG Option

If the PROGRAM=BIG option appears, the O option (see processor call options, 10.5.1) is turned on. The compiler will generate code that allows D-bank addresses to exceed octal 0200000 (decimal 65535). The use of this option will eliminate problems that result when a program that needs the O option is mistakenly compiled without it.

9. Debug Facility Statements

9.1. General

The debug facility is used during the debugging phase in the creation of a FORTRAN program, and ordinarily would not be a part of the final program unit.

Five different options provide the debugging aids: subscript checking, label tracing, tracing of changes in values, tracing of entry and exit for subprograms, and simple output.

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). The debug packet identification statement (AT) identifies the beginning of the debug packet, and the point in the program unit at which the statements in the debug packet are to be executed. The three executable statements (TRACE ON, TRACE OFF, and DISPLAY) designate actions to be taken at specific points in the program unit.

At most one DEBUG statement may appear in a program unit; if a DEBUG statement appears, then any number (including zero) of debug packets may appear (in the program unit). The AT, TRACE ON, TRACE OFF, and DISPLAY statements may not appear before the DEBUG statement. Within the program unit, debug packets must be located after all regular code of the FORTRAN main program or subprogram, but preceding the END statement (or the last statement in the program unit, if the program unit has no END statement). Any normal FORTRAN executable, data, or format statement may also occur in a debug packet. The debug packet may be terminated only by another AT statement or the END statement for that FORTRAN main program or subprogram (or by the FUNCTION or SUBROUTINE statement that signifies the start of an internal subprogram).

The debug facility and compiler optimization (V or Z option on the ASCII FORTRAN processor call statement) are incompatible. If used together, bad code may be generated.

The individual debug statements (which are all nonstandard) are explained in detail in the following subsections.

9.2. DEBUG

Purpose:

The DEBUG command indicates the existence of a debug facility for the given FORTRAN program unit and specifies the debugging environment.

Form:

```
DEBUG [option [ ,option] ... ]
```

where *option* is any of the five debugging environment specifications.

Description:

The five debugging environment *options* are:

- UNIT (See 9.2.1.)
- SUBCHK (See 9.2.2.)
- TRACE (See 9.2.3.)
- INIT (See 9.2.4.)
- SUBTRACE (See 9.2.5.)

Any combination of these five options may appear in the option list following the DEBUG keyword. They may be given in any order.

There must be a single DEBUG statement for each program unit to be debugged and it must immediately precede the first debug packet (if one exists).

If the UNIT option is not specified, any debugging output will be put in the standard program output file.

If the TRACE option is omitted from the DEBUG option list, there can be no display of program flow by statement labels within the program unit.

Examples:

```
DEBUG
C      Indicates debugging is enabled. Debug action is specified
C      in an associated AT statement. Output is put in
C      the standard system output file.
DEBUG SUBTRACE,UNIT(4),SUBCHK(ARRAY1,BUNCH2,GROUP3),INIT
C      Subscripts are checked for arrays ARRAY1, BUNCH2,
C      and GROUP3. Changes in values of all variables are
C      noted. Debug output is put on unit number 4.
DEBUG TRACE,INIT(C,LIST1,E),SUBCHK
C      Debugging will include subscript checking on all arrays,
C      list of program flow by statement label passage (assuming a
C      TRACE ON statement appears in an AT packet following the
C      DEBUG statement), and notation of changes in value of C,
C      LIST1, and E.
```


9.2.1. UNIT

Purpose:

The UNIT option designates a particular output file for debug information.

Form:

UNIT(*c*)

where *c* is an integer constant representing a file reference number.

Description:

All debugging output will go to the designated file.

The file number may not change within an executable program; for example, if the FORTRAN main program specifies UNIT(8), a subprogram called by this main program must specify UNIT(8) if it has a DEBUG statement.

If this option is not present, all debugging output will be put in the standard output file. No DEBUG UNIT message will be printed in this case.

Example:

```
          DEBUG UNIT (25)
C          Sends all debug output to the file associated with unit 25.
```

9.2.2. SUBCHK

Purpose:

The SUBCHK option checks the validity of subscripts of array elements referenced in the program unit.

Form:

SUBCHK [(*n* [, *n*] . . .)]

where each *n* is an array name.

Description:

If the list of array names is not given following the SUBCHK option, subscript checking is done for all arrays in the program unit.

The check is made by comparing the size of the array with the product of the subscripts. A message (listing the source code line number and array name) will be placed in the debug output file if an out-of-range subscript expression is encountered. The incorrect subscript will still be used in the continued program execution.

If this option is omitted, no subscript checking will be performed.

Subscript checking cannot be done for assumed-size arrays (see 2.2.2.4.1).

Examples:

```
DEBUG SUBCHK
```

```
DEBUG SUBCHK (ARRAY1, LIST2)
```

9.2.3. TRACE

Purpose:

The TRACE option indicates that statement label tracing is desired in the FORTRAN program unit in which the DEBUG statement appears.

Form:

```
TRACE
```

Description:

This option only enables label tracing.

Tracing will not actually be performed until a TRACE ON statement is encountered in the program flow. It is terminated upon encountering a TRACE OFF statement (see 9.4 and 9.5).

TRACE ON and TRACE OFF statements have no effect on a program unit in which the TRACE option has not been specified.

Example:

```
DEBUG TRACE
C           The trace debug facility is enabled.
C           It can be activated with a TRACE ON statement.
```

9.2.4. INIT

Purpose:

The INIT option traces the changes in values of variables and arrays during execution.

Form:

```
INIT [ ( m [ , m ] . . . ) ]
```

where *m* is the name of a variable or array in the program unit for which a value trace is to be performed.

Description:

If no list is given after the INIT option, a value trace is done on every variable or array in the program unit. This includes changes in value of any particular element of an array.

The value trace consists of placing, in the debug output file, a display of the variable name or array element name along with its new value each time it is assigned a value in an assignment statement, a READ statement (except a namelist READ), a DECODE statement, or an ASSIGN statement. The source code line number where the variable is set is also listed.

Example:

```
          DEBUG INIT (A,VAR1)
C          Starts debug facility and initiates trace of array
C          A and variable VAR1.
```

9.2.5. SUBTRACE

Purpose:

The SUBTRACE option indicates entrance and exit of a subprogram during program execution.

Form:

```
SUBTRACE
```

Description:

When the SUBTRACE option is included in the DEBUG statement within a function or subroutine, a trace on entrance to and exit from that subprogram is enabled.

The message 'ENTER SUBPROGRAM *s*' (where *s* is the subprogram entry point name) will be placed in the debug output file each time *s* is entered, and 'RETURN FROM SUBPROGRAM *s*' will be inserted in the file each time *s* completes execution.

Example:

```
          DEBUG SUBTRACE
```

9.3. AT

Purpose:

The AT statement identifies the beginning of a debug packet and indicates the point in the program unit at which the packet is to be activated (that is, the point at which the statements in the debug packet are to be executed).

Form:

AT s

where *s* is a statement label of an executable statement in the program unit to be debugged.

Description:

There must be one AT statement for each debug packet. Each AT statement indicates the beginning of a new debug packet. The end of the debug packet is indicated by an END statement (or by the FUNCTION or SUBROUTINE statement that signifies the start of an internal subprogram) or another AT statement.

The statements in the debug packet are executed whenever the statement associated with statement label *s* is executed in the program flow. Note that they are executed immediately prior to the execution of *s*.

Example:

```
DEBUG  
AT 100  
DISPLAY X,Y,A } debug packet  
END
```

C The DISPLAY statement is executed each time the statement
C with label 100 is executed.

9.4. TRACE ON

Purpose:

The TRACE ON statement initiates display of the flow of execution by statement label.

Form:

TRACE ON

Description:

After TRACE ON has been encountered and until the next TRACE OFF is encountered, a record of the associated statement label is placed in the debug output file each time a labeled statement is executed in the program.

TRACE ON remains in effect through any level of subprogram call or return. If the TRACE option has not been used on a DEBUG statement in a particular program unit, label trace will not occur during execution of that program unit.

TRACE ON may occur anywhere within a debug packet.

There can be no display of program flow by statement label within this program unit if the TRACE option was omitted from the DEBUG option list.

Example:

```
DEBUG TRACE, INIT(A,B)
```

```
  .  
  .
```

```
AT 104
```

```
TRACE ON
```

```
C           The flow of execution will be displayed starting at  
C           statement 104.
```

9.5. TRACE OFF

Purpose:

The TRACE OFF statement terminates statement label tracing.

Form:

TRACE OFF

Description:

TRACE OFF may occur anywhere within a debug packet. This statement terminates tracing of program flow by statement label in the program.

Example:

```
DEBUG TRACE, INIT(A,B)
```

```
AT 104  
TRACE ON
```

```
AT 950  
TRACE OFF
```

```
C      The flow of execution will be displayed from the point  
C      where statement 104 is executed to the point where statement  
C      950 is executed.
```

9.6. DISPLAY

Purpose:

The DISPLAY statement provides a simple debug output mechanism.

Form:

DISPLAY *list*

where *list* is a series of variables, array element names with constant subscripts, or array names separated by commas. A formal parameter name of a function or subroutine is not permitted in *list*.

Description:

The DISPLAY statement is equivalent to the following FORTRAN statements:

```
NAMELIST /name/ list  
WRITE (n,name)
```

where *list* is as defined above, *name* is a name generated for DISPLAY which is not a legal symbolic name, and *n* is the debug file reference number (from the UNIT option). DISPLAY provides a simple means of putting results of debugging operations for the program unit in the debug output file without needing FORMAT, NAMELIST, or WRITE statements. The output to the debug output file is in NAMELIST format. The DISPLAY statement may appear anywhere in a debug packet.

Example:

```
AT 100  
DISPLAY A,B,C,D(1,2),E  
C      The values of variables A, B, C, and E and array  
C      element D(1,2) are listed each time before  
C      statement 100 is executed.
```

9.7. Debug Facility Example

The following example uses all of the debug facility statements (including all of the DEBUG statement options). It includes a main program with no DEBUG statement, and an external subroutine (A) with a DEBUG statement and three AT packets.

FORTRAN source:

```
1.      CALL A
2.      END

3.      SUBROUTINE A
4.      DIMENSION B(4)
5.      DATA I, K /2, 5/
6. 5    B(1) = 2.
7. 10   J = B(2)
8. 15   L = B(K)
9. 20   RETURN
10.     DEBUG UNIT(6), SUBCHK, TRACE, INIT, SUBTRACE
11.     AT 5
12.     TRACE ON
13.     DISPLAY I, K
14.     AT 15
15.     TRACE OFF
16.     AT 20
17.     DISPLAY J, B
18.     END
```

Program Execution:

```
a) DEBUG UNIT          6
b) ENTER SUBPROGRAM A
c) TRACE ON
d) $0001
e) I =          2, K =          5
f) $END
g) TRACE          5
h) AT LINE          6: ELEMENT          2 OF B =          2.0000000
i) TRACE          10
j) AT LINE          7: J =          2
k) TRACE OFF
l) AT LINE          8: SUBSCRIPT IS OUT OF RANGE FOR ARRAY B
m) AT LINE          8: L =          0
n) $0002
o) J =          2,
p) B = .00000000 , .20000000+001, .00000000 , .00000000
q) $END
r) RETURN FROM SUBPROGRAM A
```


The lines printed out during the preceding execution are a result of the following options and statements:

UNIT option - line a.

SUBCHK option - line l.

TRACE option and TRACE ON, TRACE OFF, and AT statements - lines c, g, i, k.

INIT option - lines h, j, m.

SUBTRACE option - lines b, r.

DISPLAY and AT statements - lines d - f, n - q.

10. Writing a FORTRAN Program

10.1. General

This section discusses the organization of FORTRAN programs and the rules for writing them.

10.2. FORTRAN Program Organization

A FORTRAN program is made up of one, and only one, main program and as many subprograms and external procedures as required. The main program contains the primary steps required to solve a given problem. The subprograms are subordinate units used by the main program. Both are referred to as program units.

10.2.1. Program Unit

A program unit is either a main program, a subprogram, or a BLOCK DATA (specification) program. Both **internal** and external subprograms are program units. If the source input to the ASCII FORTRAN compiler consists of a main program and one or more subprograms or specification programs, then the main program must physically be the first program unit.

A program unit consists of statements and optional comment lines.

A statement is written on one or more lines. The first line is called an initial line. Succeeding lines, if any, are called continuation lines.

A comment line is not a statement. It merely provides information for documentary purposes.

10.2.2. Types of Program Units

Function subprograms and subroutine subprograms are executable subprograms and are called procedure subprograms. A subprogram headed by a BLOCK DATA statement is a nonexecutable subprogram and is called a specification subprogram.

The various types of program units, illustrated in Figure 10-1, are:

- A main program is a series of comments and statements which may begin with a PROGRAM statement, does not contain a BLOCK DATA statement, and is terminated by an END, FUNCTION, or SUBROUTINE statement.
- A function subprogram is a series of comments and statements (which does not contain a BLOCK DATA or PROGRAM statement) starting with a FUNCTION statement and terminated by an END, FUNCTION, or SUBROUTINE statement.
- A subroutine subprogram is a series of comments and statements (which does not contain a BLOCK DATA or PROGRAM statement) starting with a SUBROUTINE statement and terminated by an END, FUNCTION, or SUBROUTINE statement.
- A specification (BLOCK DATA) subprogram is a series of comments and specification statements starting with a BLOCK DATA statement and terminated by an END statement.
- A program unit is an internal subprogram if the previous program unit was terminated by the internal's FUNCTION or SUBROUTINE statement instead of an END statement.
- A program unit is external if it is a main program or a BLOCK DATA subprogram, if it is the first program unit in the compilation, or if the previous program unit was terminated by an END statement.

See 7.4.2 and 7.4.3 for descriptions and examples of functions and subroutines (both external and internal).

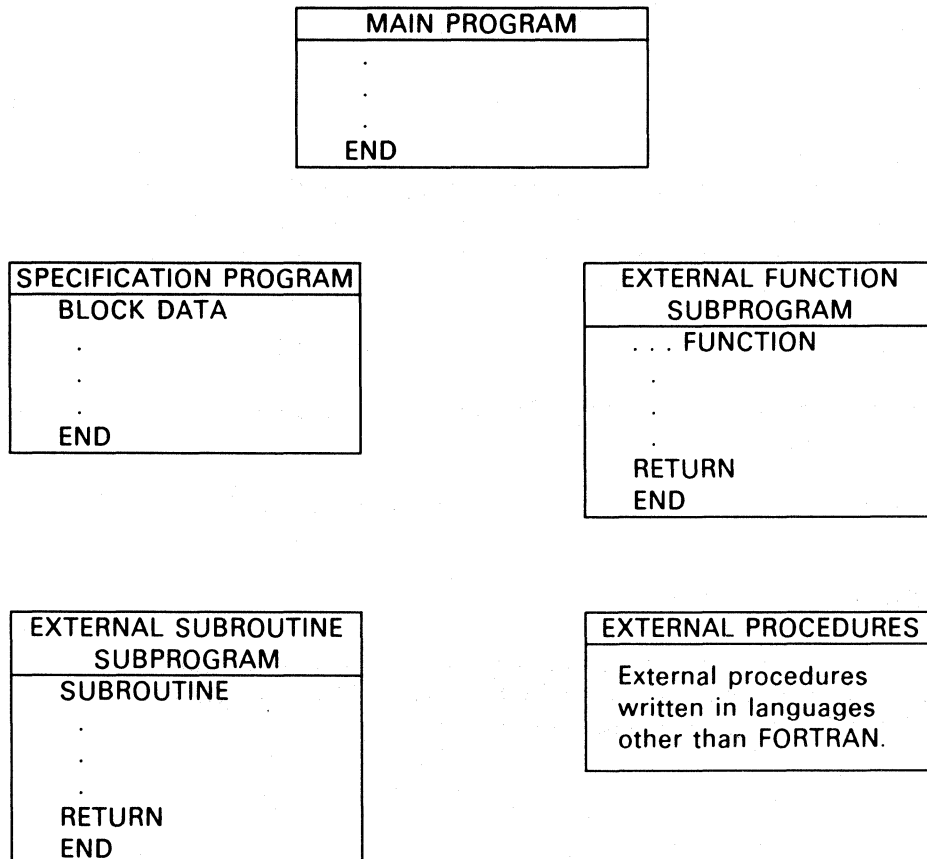


Figure 10-1. Program Units Within a FORTRAN Program

A specification program (with a BLOCK DATA header) consists entirely of nonexecutable statements and therefore never assumes control during execution.

Subprograms are useful because they eliminate repetitive coding of procedures used many times in a program. In addition, a library of mathematical external procedures, called intrinsic functions (see 7.3.1), is present which contains debugged procedures for computation of mathematical functions such as square root, sine, etc. The main program of a large FORTRAN program can be coded as a logical skeleton consisting primarily of references to subprograms; these subprograms can be independently coded and may or may not be compiled with the main program.

All compiled program units are linked together by the Series 1100 Collector to form an executable program starting with the main program unit. Program units may be written in languages other than FORTRAN but must conform to the rules for FORTRAN subprograms. (See Appendix K, Interlanguage Communication.) Such program units and procedure subprograms (function and subroutine subprograms) are termed external procedures.

10.2.3. Program Unit Organization

A program unit consists of comments and statements. A FORTRAN statement falls into one of two categories: an executable statement or nonexecutable statement. An executable statement specifies an action. A nonexecutable statement describes the characteristics and arrangement of data, editing information, statement function definitions, and classification of program units. Nonexecutable statements are generally intended as instructions to the compiler; in most cases, no executable machine language instructions are generated. Executable statements result in executable machine language instructions (object program).

Statement classification and ordering are described in 10.3. The actual formats of the lines of a FORTRAN source program are outlined in 10.4.

Note that many program units may be put together into one source element and compiled into one relocatable element.

10.2.4. Execution Sequence

Execution of a FORTRAN program begins with the performance of the first executable statement of the main program. The difference between an executable statement and a nonexecutable statement is outlined in 10.2.3.

A subprogram, when referenced by the name of the subprogram, starts execution with the first executable statement of that subprogram. When a subprogram is referenced by an entry name, execution starts with the first executable statement following the proper ENTRY statement. When subprogram activity is complete, execution of the calling program is resumed at the statement following the call, or at one of the statement labels appearing in the parameter list of the CALL statement.

The execution sequence is not affected by the appearance of nonexecutable statements or comment lines between executable statements.

Every program unit except a block data subprogram should contain at least one executable statement. A program unit should not contain a statement that can never be executed (for example, an unlabeled statement following an unconditional GO TO).

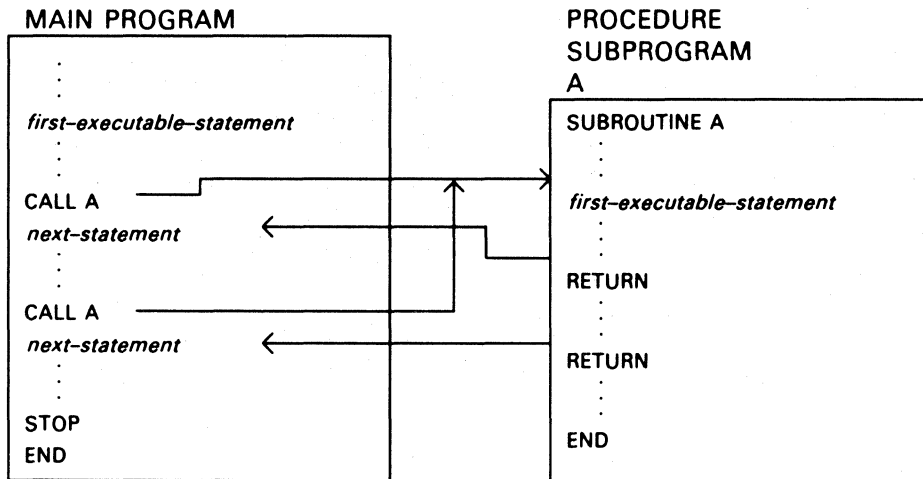
If the execution sequence attempts to proceed beyond the last executable statement of a main program, the effect is the same as the execution of a STOP statement. If the execution sequence attempts to proceed beyond the last executable statement of a procedure subprogram, the effect is the same as the execution of a RETURN statement.

In the execution of a program, a subprogram may not be referenced twice without the execution of a RETURN statement in that subprogram having intervened (that is, no recursion is allowed).

In the first example shown in Figure 10-2, the main program proceeds until it encounters a reference (CALL) to the external procedure. The external procedure assumes control until it encounters a RETURN statement, which sends control back to the calling program unit (in this case, the main program). The main program then continues processing until another reference transfers control to the external procedure. The external procedure assumes processing until it encounters a RETURN statement (not necessarily the same one as the first RETURN statement) and transfers control back to the main program. The main program then resumes processing until it encounters the STOP statement, which transfers job control to the operating system.

The second example in Figure 10-2 shows how a procedure subprogram can call upon another procedure subprogram during execution.

Example 1:



Example 2:

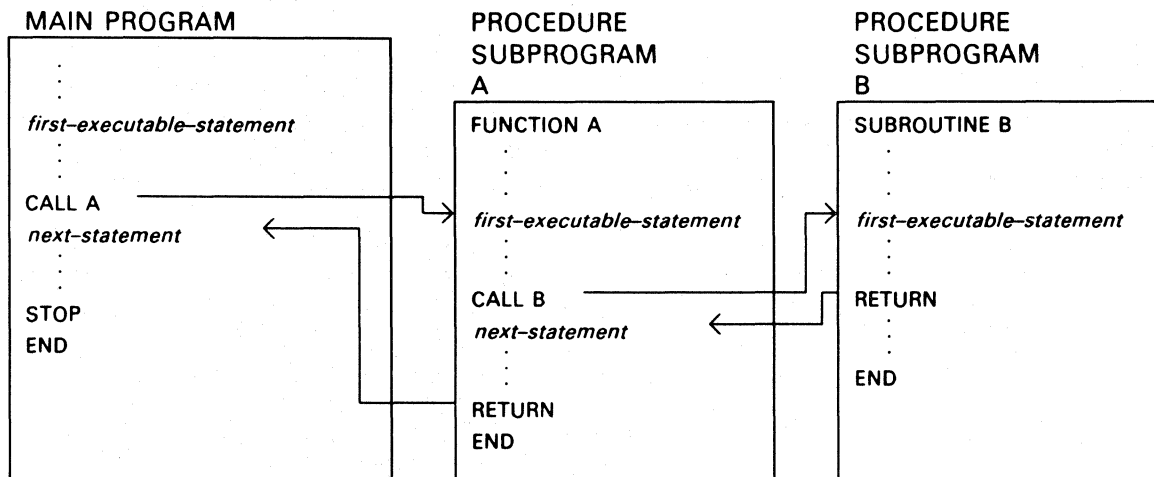


Figure 10-2. Sample Control Paths During Execution

10.3. Statement Categories

A given FORTRAN statement performs one of three functions:

- It causes certain operations to be performed (for example, addition, multiplication, branching)
- It specifies the nature of the data being handled
- It defines the characteristics of the source program

FORTRAN statements usually are composed of certain FORTRAN keywords used in conjunction with the basic elements of the language: constants, variables, and expressions. The nine categories of FORTRAN statements are:

- **Assignment statements:** These statements usually cause calculations to be performed. The result replaces the current value of a designated variable, array element, or substring.
- **Control statements:** These statements enable the user to govern the order of execution for the object program and to terminate its execution.
- **Input/output statements:** These statements, in addition to controlling input/output devices, enable the user to transfer data between internal storage and an input/output medium.
- **FORMAT statement:** This statement is used in conjunction with certain input/output statements to specify the form in which data appears in a FORTRAN record or an input/output device.
- **NAMelist statement:** This statement is used in conjunction with certain input/output statements to specify data appearing in a special kind of record.
- **DATA initialization statement:** This statement is used to assign initial values to variables and array elements.
- **Specification statements:** These statements are used to declare the properties of variables, arrays, and functions (such as type and amount of storage reserved).
- **Statement function definition statement:** This statement specifies operations to be performed whenever the statement function name appears in an executable statement.
- **Subprogram statements:** These statements enable the user to name and to specify arguments for functions and subroutines.

10.3.1. Statement Classification

Each statement is classified as executable or nonexecutable.

Executable statements specify actions and form an execution sequence in an executable program. All assignment, control and input/output statements are executable statements.

Nonexecutable statements describe characteristics, arrangement, and initial values of data, contain editing information, define statement functions, specify the class of subprograms, and specify entry points within subprograms. Nonexecutable statements are not a part of the execution sequence.

Statement labels are allowed on any statement (except continuation lines of a statement). However, only labeled executable statements (except ELSE IF, ELSE, and DEFINE FILE) and FORMAT statements may be referred to by the use of statement labels. There is one exception to this rule: the DELETE statement may refer to any statement label.

A label on a FORMAT statement cannot be used as a transfer label by another statement in the program unit (that is, it cannot be used to control the execution sequence).

See Appendix F for a breakdown of executable and nonexecutable statements.

10.3.2. Ordering of Statements and Lines

The order of a FORTRAN program unit (other than a BLOCK DATA subprogram) is:

<u>Placement</u>	<u>Statements</u>
1	COMPILER statement , if desired.
2	Subprogram (FUNCTION or SUBROUTINE) statement, if subprogram, or PROGRAM statement, if main program (optional).
3	IMPLICIT statements, if any.
4	Explicit specification statements, if any. Generally, the order is as follows: Explicit type statements, PARAMETER statements, DIMENSION statements, BANK statements , COMMON statements, EQUIVALENCE statements, EXTERNAL statements, INTRINSIC statements, SAVE statements, DATA statements, NAMELIST statements , FORMAT statements.
5	Statement function definitions, if any.
6	Executable statements, at least one of which should be present.
7	END statement (see 4.9).

Within a program unit, FORMAT, **DEFINE FILE**, and **EDIT** statements and comment lines may appear anywhere. ENTRY statements may appear anywhere except within ranges of DO-loops and block IF structures or within main programs or BLOCK DATA subprograms.

PARAMETER statements may occur before and among other specification statements. DATA statements should occur after all specification statements.

Figure 10-3 is a pictorial representation of the required order of statements for a program unit. Vertical lines delineate varieties of statements which may be interspersed. For example, FORMAT statements may be interspersed with specification statements and executable statements. Horizontal lines delineate varieties of statements which may not be interspersed. For example, specification statements may not be interspersed with statement function definitions, and statement function definitions may not be interspersed with executable statements.

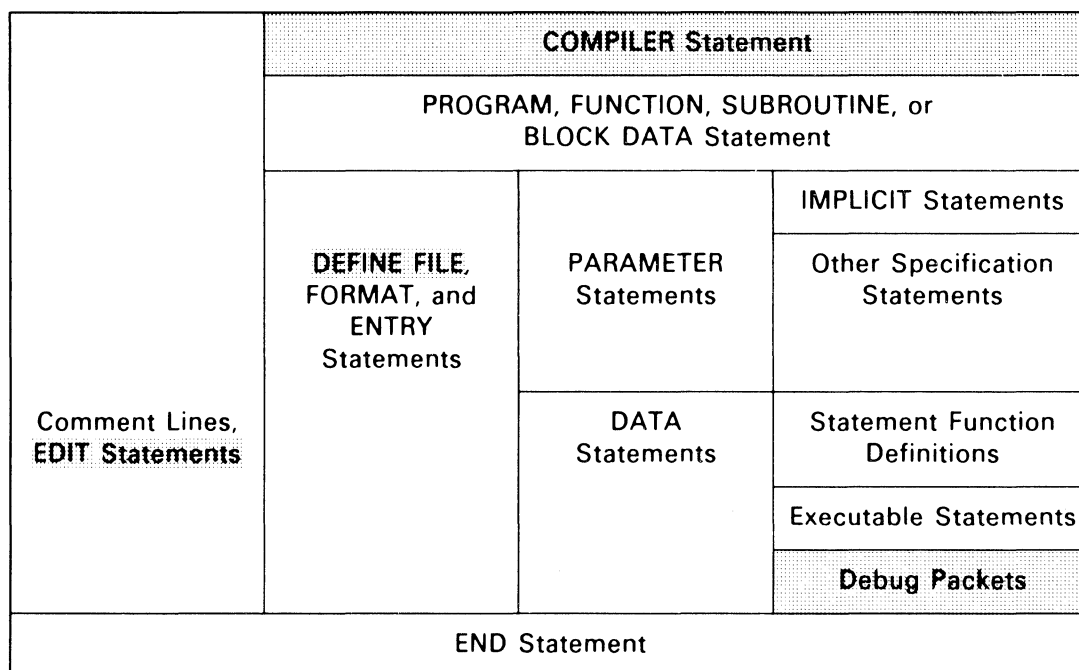


Figure 10-3. Order of Statements and Lines

10.4. Source Program Representation and Control

10.4.1. Source Program Format

A FORTRAN program consists of a series of lines. A line in a program unit is a string of 72 characters. The character positions in a line are called columns and are consecutively numbered 1, 2, 3, . . . , 72. The number indicates the sequential position of a character in the line starting at the left and proceeding to the right. Lines are ordered by the sequence in which they are presented to the processor. Thus a FORTRAN program consists of an ordered set of characters.

The following is an illustration of the two basic forms of a source program line. The first form is used for all FORTRAN statements. The appearance of a character in column 6 is used to indicate a continuation line. The second form is used for comment lines.

1	5	6	7	73	80	
number or blank					statement	nonprocessed documentation

1	2	73	80
C or *	comment		nonprocessed documentation

A FORTRAN line uses only columns 1 through 72. The information in columns 73 through 80, shown only in the program listing, may be used for documentation. For statements, columns 1 through 5 are used for labels. Column 6 is used to indicate the continuation of a statement. Columns 7 through 72 contain the statement or its continuation. If the line is a comment line, columns 2 through 72 contain the comment information.

10.4.1.1. Comment Line

An asterisk (*) or the letter "C" in column 1 of a line designates that line as a comment line. Any character capable of being represented in the processor may appear in a comment line, occupying columns 2 through 72. A comment line does not affect the program in any way and is available as a convenience for the user as a documentation/information device.

A comment line should be immediately followed by an initial line or another comment line. A comment line may precede the initial line of the first statement of any program unit.

Any line with columns 1-72 entirely blank is also considered to be a comment line.

10.4.1.2. Statements

A statement consists of an initial line and, if required, continuation lines which follow in sequence. Each line of the statement is written in columns 7 through 72. An initial line must have either the digit 0 or a blank character in column 6 and must not have the character C or * in column 1. Each continuation line must have a character other than the digit 0 or blank character in column 6, and must have blanks in columns 1 through 5. Each continuation line should be immediately preceded by an initial line or another continuation line.

Except as part of a logical IF statement, no statement can begin on a line that contains any part of the previous statement. For example,

```
IF (LOG) THEN I=I+1
```

is illegal, because a block IF statement and an assignment statement appear on the same line.

Blank characters within a statement do not change the interpretation of a statement except when they appear within the data strings of Hollerith constants, character constants, or the Hollerith or character field descriptors in FORMAT and I/O statements. Nonsignificant blank characters do not count as characters in the limit of 1320 characters in any one statement.

Blank lines, like blank columns, have no effect on the code generated. Blank lines may be used wherever desired to insert spaces in the listing of the source program. A line which is blank in columns 1 through 72 is not regarded as an initial line.

For example:

```
40 FORMAT(17H AVERAGE VALUE = , F10.5)
```

could be written as:

```
40 FORMAT(  
  A17H AVERAGE VALUE =  
  B, F10.5)
```

However, it cannot be written as:

```
40  FORMAT(17H
      AVERAGE VALUE =
      B, F10.5)
```

because the 17H of the FORMAT statement means that the 17 characters immediately following the H are Hollerith data. Since blank characters are significant in Hollerith data, the next 17 blanks are interpreted to be the data value rather than the intended characters.

10.4.1.3. Statement Labels

A statement label is an unsigned integer (1 through 99999) that identifies a FORTRAN statement and is written in columns 1 through 5 of the initial line of a statement. Only the digits 0-9 and blank characters may be used in a statement label. The same statement label cannot be used more than once in a program unit, but the same statement label can appear in more than one program unit. The value of a statement label does not affect the order in which statements are executed, it merely identifies the statement it is associated with. This enables other statements of the program unit to reference it. See 10.3.1 for rules stating which statement labels may be referenced.

A maximum of five digits may be used in a statement label. All blanks and leading zeros are ignored.

For example, in the sequence:

```
      .
      .
157 F = (A + B - C) / D
      .
      .
      GO TO 157
      .
      .
```

the statement label of the arithmetic assignment statement could have been written as:

```
1Δ5Δ7 F = (A + B - C) / D
```

or:

```
01Δ57 F = (A + B - C) / D
```

or:

```
15ΔΔ7 F = (A + B - C) / D
```

but not as:

```
157Δ0 F = (A + B - C) / D
```

because this label is 1570 rather than 157.

10.4.2. Compilation Listing

For every program compiled, a compilation listing is produced. The composition of the information listed is a function of the listing options specified by the user.

10.4.2.1. Listing Options

The options available to control the composition of the compiler listing are specified in the @FTN processor call command (see 10.5) and in EDIT statements. The processor options are used to specify the initial setting for the type of listing desired. They may be overridden at any point in the source program by the EDIT statement (see 8.4).

All compiler options, including nonlisting options, are listed in 10.5.1. The options that may be specified on the @FTN processor call to control the compiler listing are:

<u>Option</u>	<u>Meaning</u>
D	A listing of the storage map, common blocks, entry points, and external references is generated in addition to the identification line and terminal message line.
K	If the source lines are being printed (S or L option), then both the update and input (base) line numbers will appear in front of the source images. In addition, SIR correction lines will be interspersed within the source code listing.
L	In addition to the information listed for the S option, the cross reference listing, storage map, common block list, entry point list, external reference list, and octal and mnemonic representation of the generated object code are also to be printed.
N	If used with no other listing options, a minimum of printout is produced. This output consists of a single compiler identification line with time and date, and a single line indicating the end of compilation with the number of errors and warnings, and the amount of storage for I-bank, D-bank, and common blocks. If any errors are detected during the compilation, error messages are printed. Warnings are not printed. See 10.10 for information on the format of error messages. If used with other listing options, the N option causes warnings to be suppressed.
R	A cross-reference listing is to be generated in addition to the identification line and terminal message line.
S	A listing of the source program statements is to be produced in addition to the specifications provided by the N option. All warnings detected throughout the compilation are printed in addition to any error messages. Errors or warnings detected during syntax analysis may be interspersed within the source program listing and are usually printed immediately following the offending statement. See 10.10 for further details on diagnostics.
T	Print diagnostics for nonstandard FORTRAN usage (see 10.5.1).
W	The correction lines used by the source input routine (SIR) are to be printed at the top of the listing (before the source code listing, if any).
Y	A listing containing the storage map, common blocks, entry points, external references, and a mnemonic representation of the object code is generated.

10.4.2.2. Composition of a Compilation Listing

The composition of a listing generated under control of the L option is described in this subsection. Refer to 10.4.2.1 to determine which parts of the listing described are not generated under control of the other listing options.

See Figure 10-4 for an example of an L option listing.

10.4.2.2.1. Identification Line

The identification line printed at the beginning of each compilation listing has the following format:

```
FTN ss Rii - mm/dd/yy - hh:mm - (i,o)
```

where:

FTN is the ASCII FORTRAN compiler used to perform the compilation.

ss Rii is the level number of the compiler; *ss* denotes the symbolic update number; *ii* denotes the incremental update number. Following *ii*, a letter may be used to indicate minor modifications, such as emergency fixes, recollections to include modified RLIB\$ elements, etc. (for example, 9R1A). Release levels begin with 1R1.

mm/dd/yy indicates the month, day, and year on which the compilation was performed.

hh:mm is the hour and minute at which the compilation began.

(*i,o*) indicates the input cycle number and the output cycle number, respectively. If *i* is blank, the input was from the runstream. If *o* is blank, there is no symbolic output element.

10.4.2.2.2. Source Code Listing

The source code listing contains all source language input lines processed by the compiler. The errors detected during the production of this listing are printed on the first available line following the offending source line.

The example in Figure 10-4 shows a sample source code listing with a main program, an internal subroutine in the main program, an external subroutine subprogram, and an internal subroutine in that external subroutine.

The first (leftmost) field of the listing contains the level number of the statement (if greater than zero). The level number is a counter containing the DO-level plus the IF-level of the statement. The DO-level is a counter containing the number of nested DO-loops; the IF-level is a counter containing the number of nested block IF structures.

The level number column will contain blanks, if the level number is zero, and the letter D if the source line being printed has been deleted by the DELETE statement.

Note that since the example contains no block IF or DO statements or deleted lines, the level number column is always left blank.

```
@FTN,L A.TEST,B.
FTN 10R1 04/21/81-08:11(1.)
  1.      I = 1
  2.      PRINT *,@MAIN: I = @ , I
  3.      CALL INTMN
  4.      CALL SUB1
  5.      C
  6.      C *** INTERNAL SUBPROGRAM (INTMN) IN MAIN PROGRAM ***
  7.      C
  8.      SUBROUTINE INTMN
  9.      I = 2
 10.      PRINT *,@INTMN: I = @ , I
 11.      RETURN
 12.      END
 13.      SUBROUTINE SUB1
 14.      COMMON I
 15.      I = 3
 16.      PRINT *,@SUB1: I = @ , I
 17.      CALL INTSUB
 18.      RETURN
 19.      C
 20.      C *** INTERNAL SUBPROGRAM (INTSUB) IN EXTERNAL SUBPROGRAM (SUB1) ***
 21.      C
 22.      SUBROUTINE INTSUB
 23.      INTEGER I
 24.      COMMON /C/ I
 25.      I = 4
 26.      PRINT *,@INTSUB: I = @ , I
 27.      RETURN
 28.      END
```

FORTRAN CROSS REFERENCE LISTING

MAIN PROGRAM

NAME	USE	LINE NUMBER
I	SET	1
	USED	2
INTMN	USED	3
SUB1	USED	4

Figure 10-4. L Option Listing (Part 1 of 5)

SUBROUTINE INTMN : MAIN PROGRAM

NAME	USE	LINE NUMBER
I	SET	9
	USED	10
INTMN	SPEC	8

SUBROUTINE SUB1

NAME	USE	LINE NUMBER
I	COMMON	14
	SET	15
	USED	16
INTSUB	USED	17
SUB1	SPEC	13

SUBROUTINE INTSUB : SUB1

NAME	USE	LINE NUMBER
C	SPEC	24
I	COMMON	24
	SPEC	23
	SET	25
	USED	26
INTSUB	SPEC	22

F O R T R A N O B J E C T C O D E L I S T I N G

MAIN PROGRAM

RELATIVE ADDRESS	INSTRUCTION F J A X H I U	SUBFIELDS	U FLD LC	LABEL	SYMBOLIC F, J A, U, X	INSTRUCTION	LINE NUMBER
				\$(1)	AXR\$		
					ASCII		
000000		00000000000000X	2	\$(6)	+	00000000000000	0
000000	74	13 13 00 0 000000X	4	\$(1)	LMJ	X11, FINT2\$	0
000001	10	16 04 00 000001			LA, U	A4, 1	1
000002	01	00 04 00 0 000000R	0		SA	A4, 1	1
000000		07777777777776		\$(4)	+	07777777777776	2
000001		00304000000000			+	00304000000000	2
000002		00000000000000			+	00000000000000	2
000003		00000000000000			+	00000000000000	2
000004		00000000000000R	10		+	00000000000000	2

Figure 10-4. L Option Listing (Part 2 of 5)

RELATIVE ADDRESS	INSTRUCTION F	SUBFIELDS J A X HI U	U FLD LC	LABEL	SYMBOLIC F, J	INSTRUCTION A, U, X	LINE NUMBER
000005		006001200000			+	006001200000	2
000006		000000000000R	0		+	000000000000	2
000007		001000000000			+	001000000000	2
000010	74	13 13 00 0 00000X	5		LMJ	X11, FMTE\$\$	2
000003	26	16 14 00 000000 R	4	\$(1)	LXM, U	AO, FTEMP\$	2
000004	46	17 13 00 000000 X	1		LXI, XU	X11, BDICALL\$+F2SE\$\$	2
000005	00	00 13 00 0 00000X	6		IBJ\$	X11, F2SE\$\$	2
000006	10	16 00 00 000000			LA, U	AO, 0	3
000007	46	17 13 00 000000			LXI, XU	X11, 0	3
000010	74	13 13 00 0 000017R	1		LMJ	X11, INTMN	3
000011	10	16 00 00 000000			LA, U	AO, 0	4
000012	46	17 13 00 000000			LXI, XU	X11, 0	4
000013	74	13 13 00 0 00000X	7		LMJ	X11, SUB1	4
000014	74	13 13 00 0 00000X	8		LMJ	X11, FEXIT\$	8
000000		0115101111116		\$(10)	+	0115101111116	8
000001		0072040111040			+	0072040111040	8
000002		0075040040040			+	0075040040040	8
SUBROUTINE INTMN : MAIN PROGRAM							
000015	27	00 13 00 0 000011R	4		AXR\$		
000016	74	04 00 13 0 00000			ASC II		
000017	06	00 13 00 0 000011R	4	INTMN	LX	X11, FTEMP\$+9	8
000020	10	16 04 00 000002		OG	J	0, X11	8
000021	01	00 04 00 0 000000R	0		SX	X11, FTEMP\$+9	8
000012		0777777777776		\$(4)	LA, U	A4, 2	9
000013		0030400000000			SA	A4, 1	9
000014		0000000000000			+	0777777777776	10
000015		0000000000000			+	0030400000000	10
000015		0000000000000			+	0000000000000	10
000015		0000000000000			+	0000000000000	10
000016		0000000000003R	10		+	0000000000003	10
000017		0060013000000			+	0060013000000	10
000020		0000000000000R	0		+	0000000000000	10
000021		0010000000000			+	0010000000000	10
000022	74	13 13 00 0 00000X	5		LMJ	X11, FMTE\$\$	10
000022	26	16 14 00 000012 R	4	\$(1)	LXM, U	AO, FTEMP\$+10	10
000023	46	17 13 00 000000 X	1		LXI, XU	X11, BDICALL\$+F2SE\$\$	10
000024	00	00 13 00 0 00000X	6		IBJ\$	X11, F2SE\$\$	10
000025	74	04 00 00 0 000015R	1		J	\$-8	11
000003		0111116124115		\$(10)	+	0111116124115	12
000004		0116072040111			+	0116072040111	12
000005		0040075040040			+	0040075040040	12

Figure 10-4. L Option Listing (Part 3 of 5)

SUBROUTINE SUB1

RELATIVE ADDRESS	INSTRUCTION F	J	A	X	HI	U	U FLD LC	LABEL	SYMBOLIC F, J	INSTRUCTION A, U, X	LINE NUMBER
								\$(1)	AXR\$		
									ASCII		
000026	27	00	13	00	0	000023R	4		LX	X11, FTEMP\$+19	12
000027	74	04	00	13	0	000000			J	O, X11	12
000030	06	00	13	00	0	000023R	4	SUB1	SX	X11, FTEMP\$+19	12
000031	10	16	04	00	000003			1G	LA, U	A4, 3	15
000032	01	00	04	00	0	000000R	2		SA	A4, I	15
000024						077777777776		\$(4)	+	077777777776	16
000025						0030400000000			+	0030400000000	16
000026						0000000000000			+	0000000000000	16
000027						0000000000000			+	0000000000000	16
000030						0000000000006R	10		+	0000000000006	16
000031						0060012000000			+	0060012000000	16
000032						0000000000000R	2		+	0000000000000	16
000033						0010000000000			+	0010000000000	16
000034	74	13	13	00	0	000000X	5		LMJ	X11, FMTE\$\$	16
000033	26	16	14	00	000024	R	4	\$(1)	LXM, U	AO, FTEMP\$+20	16
000034	46	17	13	00	000000	X	1		LXI, XU	X11, BDICALL\$+F2SE\$\$	16
000035	00	00	13	00	0	000000X	6		IBJ\$	X11, F2SE\$\$	16
000036	10	16	00	00	000000				LA, U	AO, O	17
000037	46	17	13	00	000000				LXI, XU	X11, O	17
000040	74	13	13	00	0	000044R	1		LMJ	X11, INTSUB	17
000041	74	04	00	00	0	000026R	1		J	\$-11	18
000006						0123125102061		\$(10)	+	0123125102061	22
000007						0072040111040			+	0072040111040	22
000010						0075040040040			+	0075040040040	22

SUBROUTINE INTSUB : SUB1

RELATIVE ADDRESS	INSTRUCTION F	J	A	X	HI	U	U FLD LC	LABEL	SYMBOLIC F, J	INSTRUCTION A, U, X	LINE NUMBER
								\$(1)	AXR\$		
									ASCII		
000042	27	00	13	00	0	000035R	4		LX	X11, FTEMP\$+29	22
000043	74	04	00	13	0	000000			J	O, X11	22
000044	06	00	13	00	0	000035R	4	INTSUB	SX	X11, FTEMP\$+29	22
000045	10	16	04	00	000004			2G	LA, U	A4, 4	25
000046	01	00	04	00	0	000000R	11		SA	A4, I	25

Figure 10-4. L Option Listing (Part 4 of 5)

```

000036      077777777776      $(4)      +      077777777776      26
000037      0030400000000      +      0030400000000      26
000040      0000000000000      +      0000000000000      26
000041      0000000000000      +      0000000000000      26
000042      0000000000011R    10      +      0000000000011      26
000043      0060014000000      +      0060014000000      26
000044      0000000000000R    11      +      0000000000000      26
000045      0010000000000      +      0010000000000      26
000046  74  13  13  00  0  000000X  5      LMJ      X11,FMTE$$      26
000047  26  16  14  00  000036  R  4      $(1)     LXM,U      AO,FTMP$+30      26
000050  46  17  13  00  000000  X  1      LXI,XU     X11,BDICALL$+F2SE$$      26
000051  00  00  13  00  0  000000X  6      IBJ$      X11,F2SE$$      26
000052  74  04  00  00  0  000042R  1      J          $-8          27
000011      0111116124123      $(10)     +      0111116124123      28
000012      0125102072040      +      0125102072040      28
000013      0111040075040      +      0111040075040      28
END

```

F O R T R A N S T O R A G E M A P

NAME	TYPE	MODE	RELATIVE ADDRESS	LOC COUNT	ELEMENT LENGTH	NUMBER OF ELEMENTS	COMMON SIZE	PROGRAM UNIT
I	INTEGER	SCALAR	000000	0	4			MAIN PROGRAM
I	INTEGER	SCALAR	000000	2	4			SUBROUTINE SUB1
I	INTEGER	SCALAR	000000	11	4			SUBROUTINE INTSUB : SUB1
COMMON BLOCKS								
C		COMMON	000000	11			1	
	BLANK	COMMON	000000	2			1	
ENTRY POINTS								
FMAIN\$		ENTRY	000000	1				
SUB1		ENTRY	000030	1				
EXTERNAL REFERENCES								
IBJ\$	BDICALL\$	BDIREF\$	FMAIN\$	FINIT\$	FMTE\$\$	F2SE\$\$	SUB1	FEXIT\$

END FTN 43 IBANK 53 DBANK 2 COMMON

Figure 10-4. L Option Listing (Part 5 of 5)

The second field of this listing is the source line sequence number. The line sequence numbers are consecutive integers, one per line, followed by a period. There is one exception: if the source element was created by CTS (see CTS Programmer Reference, UP-7940 (see Preface)), then the CTS line number will appear in the second field.

If the K option is specified, then the update line number will be followed by the input (base) line number. This option may be useful if SIR correction lines are used.

The final field is the FORTRAN source line.

If the source input to the compiler consists of more than one external program unit, then spacing is inserted before the source code listing of the next external program unit.

10.4.2.2.3. Cross Reference Listing

The cross reference listing contains all references to statement labels, subprograms, and variables which appear in the source program. Statement numbers appear first in the listing and are in numerical order. Symbolic names (variables, arrays, etc.) appear second and are in alphabetical order.

A separate cross reference listing is generated for each program unit. A heading appears before each cross reference listing, identifying the program unit. The heading has one of the following formats:

MAIN PROGRAM [*p*]

SUBROUTINE *n* [:*e*]

FUNCTION *n* [:*e*]

BLOCK DATA *b*

BLOCK DATA (DEFINED AT LINE *m*)

In these formats, *p* indicates the optional program name, *n* indicates the subprogram name, *e* (specified only if *n* is an internal subprogram) indicates the name of the external program unit (MAIN PROGRAM or external subprogram name), *b* indicates the BLOCK DATA program name, and *m* indicates the source line number where the unnamed BLOCK DATA program unit begins.

Column headings printed at the top of each page of the listing are:

<u>Column Heading</u>	<u>Significance</u>
NAME	Entity named in the program.
USE	Indication as to whether the entity was set, defined, specified, equivalenced, or used in a COMMON statement.
LINE NUMBER	The line numbers of the source statements where the entity was referenced. If an item appears in a specification statement within a FORTRAN procedure (named in an INCLUDE statement), then the line number is printed as <i>proc-name .proc-line-number</i> .

10.4.2.2.4. Object Code Listing

The L-option listing produces an object program listing containing all instructions, data and constants generated by the compiler for the program.

A heading appears before the object code listing of each program unit, identifying the unit. This heading is the same as that generated for the cross reference listing.

At the top of each page of this listing, column headings are printed which refer to the fields under them:

<u>Column Heading</u>	<u>Significance</u>
RELATIVE ADDRESS	The relative address, in octal, of the memory location.
INSTRUCTION SUBFIELDS	Machine representation for F, J, A, X, H, I, and U instruction subfields. Letters R and X are also printed for the relocation information and the external reference information, respectively. This portion is suppressed, except the R and X, when the Y option is specified. For a detailed explanation of each subfield, see the AIM Supplementary Reference, UP-9047 (see Preface).
U FLD LC	The location counter number to be applied to the U-field for relocation purposes (if R appears under INSTRUCTION SUBFIELDS), or the index to the external reference table to be applied to the U-field for external reference purposes (if X appears under INSTRUCTION SUBFIELDS).
LABEL	Shows the label associated with the storage location, if any. This label is composed of a decimal number and the letter L or G following the number. The letter L is used for a user-defined label, while the letter G is used for a compiler-generated label.
SYMBOLIC INSTRUCTION	Contains the symbolic representation of the object code shown under INSTRUCTION SUBFIELDS. It follows assembly language code conventions as closely as possible, with the F and J fields followed by the A, U, and X fields.
LINE NUMBER	Contains the source code line number that causes the instruction to be generated. Instructions moved by optimization may be associated with a previous statement label.

10.4.2.2.5. Storage Assignment Map

A storage map of entities defined in the source program is produced after the object code listings. Statement labels appear first in the listing and are in numerical order. Symbolic names (variables, arrays, etc.) appear next and are in alphabetical order.

A single storage map is generated for all program units.

At the top of each listing, a heading line identifying the fields of each line in the listing is printed:

<u>Column Heading</u>	<u>Significance</u>
NAME	Name used for the entity defined in the source program.
TYPE	An indication of what the entity identifies. Data entities are: integer, real, character, logical, etc.
MODE	An indication of the way the entity is used (that is, scalar, array, etc.).
RELATIVE ADDRESS	Relative address (within the location counter) of the identifier. 'DUMMY' if the item is a dummy argument or character function entry point name.
OFFSET	Byte number where a character item begins (in the word defined by the relative address field). 0-Q1, 1-Q2, 2-Q3, 3-Q4.
LOC COUNT	Location counter under which the symbol is allocated. (Not used for dummy arguments.)
ELEMENT LENGTH	Element length of the entity in bytes.
NUMBER OF ELEMENTS	Number of elements that an array entity contains.
COMMON SIZE	Size of the common block in words. (Used only for the common block listing, see 10.4.2.2.6).
PROGRAM UNIT	Program unit the name is defined in.

10.4.2.2.6. Common Block Listing

Following the storage map, the heading COMMON BLOCKS is printed. A list of all common blocks referenced in the source program is then printed.

The storage map column headings are used for the common block listing. TYPE is BLANK for blank common. Otherwise, NAME identifies the name of the common block.

Only one common block listing is generated, even if the source program contains more than one program unit.

10.4.2.2.7. Entry Point Listing

Following the common block listing, a heading ENTRY POINTS is printed. The list of entry points to the source program is then printed in the format NAME, MODE (always ENTRY), RELATIVE ADDRESS, and LOC COUNT. These are similar to the entities of the storage map listing.

Only one entry point listing is generated, even if the source program contains more than one program unit.

10.4.2.2.8. External References

The external reference listing follows the entry point listing. Under the printed heading EXTERNAL REFERENCES, the names of all external entry points referenced in the source program are listed. These include the names of FORTRAN subprograms and all the library and miscellaneous routines referenced.

Only one external reference listing is generated, even if the source program contains more than one program unit.

10.4.2.2.9. Termination Message

At the conclusion of the listing, a termination line is printed. On this line, the total number of errors and warnings detected in the program and the amount of storage for I-bank, D-bank, and common blocks are printed. All fields are optional. If any field is zero, it is not printed. The format is:

```
| END FTN [n ERRORS][m WARNINGS][i NON-STD USAGES][x IBANK][y DBANK][z COMMON]
```

where:

n is the number of errors detected.

m is the number of warnings detected.

| *i* is the number of nonstandard usage messages printed (T option only).

x indicates the number of words of storage used in the instruction bank.

y indicates the number of words of storage used in the data bank, except for common blocks.

z indicates the number of words of storage used in common blocks.

10.5. Calling the ASCII FORTRAN Processor

The processor call used to initiate the ASCII FORTRAN compiler is:

```
@FTN [ ,option[option] ...] [si][ , [ro][ , so]]
```

where:

- option* is a processor call option letter. These are described in Table 10-1. Options identified as SIR options control the Source Input Routine. The listing options are discussed in more detail in 10.4.2.1.
- si* is the program file symbolic element which contains the program to be compiled if the I option is not specified. If *si* is not specified, the FORTRAN source program must follow the processor call in the runstream. If the I option is specified, the FORTRAN source program must follow the processor call in the runstream. If *si* is present with I, it designates the program file symbolic element in which the source program will be saved.
- ro* is the program file relocatable element which will receive the compiled program. If *ro* is omitted, the file name and element name of *si* are used. If *si* is also omitted, the temporary file TPF\$.NAME\$ is used.
- so* is the program file symbolic element in which the updated source program will be placed. *so* is not used if the I or U option is specified or if *si* is not specified. If neither *so* nor U is specified and there are corrections, the corrected source is compiled but not saved.

Additional control over the compilation process can be obtained by using the COMPILER statement. (See 8.5.)

10.5.1. Processor Call Options

ASCII FORTRAN processor call options are listed in Table 10-1. The listing options are described in detail in 10.4.2.1.

Table 10-1. Processor Option Letters

Letter	Action
A	Do not enter ERR mode on serious errors (that is, always perform ER EXIT\$ to terminate the compilation). Without the A option, certain errors (such as I/O errors) result in ERR\$ termination.
B	In checkout mode, this option inhibits clearing of core before loading the user program.
C	Invokes FORTRAN checkout mode, generating code in core and executing it. (See 10.6.)
D	Print the storage map, common block listing, entry point listing, and external reference listing.
E	1110 code reordering. See 8.5.4 on the COMPILER (U1110=OPT) statement.
F	Generate diagnostic tables under location counter 3 for use by walkback and the interactive PMD (see 10.7).
G	SIR option; input is compressed symbolic in columns 1-80. (Presently inactive.)
H	SIR option; source input lines may not be scanned past column 72. This option is not significant for ASCII FORTRAN. (Presently inactive.)
I	SIR option; source input is from the runstream. The source element specified in the <i>si</i> field of the processor call statement will be ASCII, unless the P option is also specified.
J	SIR option; input is compressed symbolic in columns 1-72. (Presently inactive.)
K	List update and input (base) line numbers (if source lines are being printed), and print SIR correction lines.
L	Generate all available listing output.
M	Search file FTN\$PF first for INCLUDE statements.
N	If used with no other listing options, then list only source program error messages. If used with other listing options, then suppress printing of warning messages.
O	Generate code which allows D-bank addresses to exceed octal 0200000 (decimal 65536).
P	SIR option; output symbolic in Fieldata.
Q	SIR option; output symbolic in ASCII.
R	Print the cross-reference listing.

Table 10-1. Processor Option Letters (continued)

Letter	Action
S	List only the source program and error and warning messages.
T	Flag items that are nonstandard (that is, not compatible with the American National Standard Programming Language FORTRAN, ANSI X3.9-1978) with a diagnostic. N option does not turn off these nonstandard usage messages.
U	SIR option; generate an updated cycle of the source input element.
V	Perform local code optimization. This option is ignored during checkout mode.
W	SIR option; list all SIR correction lines at the head of the printer listing.
X	Perform an ER EABT\$ at the end of the compilation if the FORTRAN program had any errors or if any serious errors (such as I/O errors) occurred.
Y	List the generated code in pseudo-assembler format, without the octal representation of each word. Also print the listings generated by the D option. The L option overrides the Y option.
Z	Perform full optimization (when not in checkout mode). This includes the operations controlled by the V option. In checkout mode, this option turns on interactive debugging (see 10.6).

10.5.2. Execution of the Object Program

10.5.2.1. Execution Using Checkout

If a program or a portion of it is in one source element, it may be executed in checkout mode by putting the C or CZ options on the compiler call (see Table 10-1). If only a subprogram is to be tested, use the CZ options. Then, upon entering checkout debug mode, test the subprogram via the CALL and DUMP commands (see 10.6).

Note the extensive debugging facilities which are available in interactive checkout debug mode.

10.5.2.2. Collection and Execution

If there are multiple source elements, or if the absolute is to be executed more than once, collect an absolute with the Collector (@MAP processor). First the FORTRAN programs must be compiled.

Next, the user must collect his program. If the local site has the FORTRAN run-time library in the SYS\$*RLIB\$ file, the user may do the following:

```
@MAP,SI MAPSOURCE,MYABSOLUTE  
IN TALLEY  
@XQT MYABSOLUTE
```

This assumes that all of the user's programs are in the standard temporary file TPF\$.

Note that a simple @XQT will force a collection before execution.

If the local site does not have the FORTRAN run-time elements in SYS\$*RLIB\$, the user must specify the appropriate file in the Collector symbolic via a LIB statement.

```
@MAP,FSI MAPSOURCE,TALLEY  
LIB FTN*LIB.  
IN TALLEY  
@XQT TALLEY
```

Note that a site may have a type1 or type2 library for ASCII FORTRAN. A type1 library contains I/O routines which are accessed by an LMJ linkage from the ASCII FORTRAN-generated code. A type2 library contains no I/O routines; the I/O routines are in system common banks which are accessed by an LIJ linkage from compiler-generated code. The ASCII FORTRAN compiler generates an IBJ\$ instruction linkage to all I/O routines; when this IBJ\$ instruction mechanism is used, the Collector decides whether to resolve the IBJ\$ to an LMJ or LIJ linkage, depending on the type of library that is specified in the LIB statement in the collection.

If the Collector generates truncation error messages during the collection, the program is too large to fit into the standard 65K addressing space. In this case, use the O option on all ASCII FORTRAN compilations. See Appendix H for a description of large programs and multibanking.

For collected absolutes, the debugging features of the debug facility statements (see Section 9) and the run-time interactive PMD and walkback are available (see 10.7).

10.6. FORTRAN Checkout Mode

The ASCII FORTRAN compiler also can be used as a compile-and-go processor by invoking the checkout mode of operation. Adding the C option to the processor call command directs the compiler to generate code into main storage and immediately execute it when compilation is complete. This mode results in increased throughput in cases where the object program is to be executed only once and when execution is relatively short. No relocatable element is produced.

FORTRAN checkout mode also provides a powerful interactive debugging system which is enabled through the use of the Z option in addition to the C option. This combined usage of options allows the user to trace the execution, halt the execution, dump variable values, and perform other debugging activities.

Since optimization is not practical for programs that are executed only once and since it interferes with interactive debugging, the optimization features of ASCII FORTRAN are disabled in checkout mode.

Although no relocatable element is produced, a write-enabled RO file (or SI file if no RO is specified) must be available, since an omnibus element is produced under certain circumstances when in checkout mode.

10.6.1. Calling Checkout Mode

The user enters a checkout run in much the same manner as a normal compilation. However, special procedures must be followed to enter data for the executing program or to invoke the debugging features provided. A simple execution would be entered as follows:

```
@FTN,NCI file .element  
user program  
@EOF  
user data images
```

The *user program* consists of a main program (which must physically be first) and any necessary subprograms (internal and external subroutines, internal and external functions, or block data programs). An END statement (see 4.9) must terminate each program unit group. As in noncheckout mode, the only variables shared between external program units are arguments passed between program units and variables defined in common blocks.

On encountering the @EOF, all program units would be processed as one program, and execution would be initiated following compilation. The data images following the @EOF would be read and used, as necessary. The user program may, of course, be read in from a file by omitting the I option. The @EOF is still required, however, preceding the data images in order to indicate that no corrections are to be applied to the file.

If the Z option is used to enable debugging, the following commands should be entered:

```
@FTN,NCIZ file .element  
user program  
@EOF  
GO  
user data images
```

The GO is required since the debugging routine allows entry of debugging commands immediately preceding the execution of the program. These commands are entered before the GO command.

10.6.2. Interactive Debug Mode in the Checkout Compiler

10.6.2.1. Entering Interactive Debug Mode

Interactive debug mode in the checkout compiler may be entered only when the Z option is specified on the checkout call card (that is, @FTN,CZ). It is entered:

- Before the first executable statement in the FORTRAN program. Debug mode is automatically entered at this point.
- When a contingency interrupt occurs during execution of the FORTRAN program (see 10.6.4).
- When the FORTRAN program executes the statement CALL PAUSE. PAUSE has no parameters.
- When execution of the FORTRAN program has reached the END statement of the main program (that is, just before termination of the program).

Entry into debug mode at this point allows the user to execute debug commands to do the following: dump the final values of variables (see DUMP command, 10.6.3.4), restore execution to a previous state (see RESTORE command, 10.6.3.11), or dump the final contents of the program (see SNAP command, 10.6.3.15).

The message:

```
END PROGRAM EXECUTION
```

is printed before debug mode is entered.

- When the special RESTART processor (FTNR, see 10.6.6) is invoked to reenter a previous debugging session. For example:

```
@FTNR ROFILE.MYPROG
```

- When a breakpoint interrupt occurred in the previous FORTRAN statement. The SETBP command (see 10.6.3.14) is used to set the breakpoint register, which causes hardware breakpoint interrupts. The message:

```
SETBP BREAK AT LINE n
```

is printed on entry to debug mode, where *n* is the current source line number.

- When a step break has been set at the current statement using the STEP command (see 10.6.3.16). The message:

```
STEP BREAK AT LINE n
```

is printed on entry to debug mode, where *n* is the current source line number.

- When a line number break has been set at the current statement using the BREAK command (see 10.6.3.1). The message:

```
BREAK AT LINE n
```

is printed on entry to debug mode, where *n* is the current source line number.

- When a statement label break has been set at the current statement using the BREAK command (see 10.6.3.1). The message:

LABEL BREAK AT *n*L

is printed on entry to debug mode, where *n* is the statement label associated with the current statement. Another line follows the above message, stating which program unit in the FORTRAN program contains the label.

- When the subprogram called by the CALL command (see 10.6.3.2) returns. The message:

ENTER DEBUG MODE (RETURN FROM CALL COMMAND)

is printed on entry to debug mode.

For the first five cases, the message:

ENTER DEBUG MODE AT LINE *n*

is printed on entry to debug mode. In the message, *n* is the source line number of the statement where execution in the FORTRAN program was interrupted.

10.6.2.2. Soliciting Input

When the checkout compiler is in interactive debug mode, commands are solicited with ER ATREAD\$. The solicitation message is:

C:

If the command is read from an @ADD stream, if the @FTN program began execution in @BRKPT mode, or if the program is executing in batch mode, then the command image will be printed.

To leave debug mode, the GO, EXIT, and CALL commands are used. The debug commands are discussed individually in 10.6.3.

10.6.3. Debug Commands

All debug command names may be abbreviated to one letter (the initial one), except for the following (with their abbreviations in parentheses): CALL (CA), LINE (LIN), SAVE (SA), SETBP (SETB), SNAP (SN), and STEP (ST).

The following syntax rules apply for the debug commands:

- No blank characters are allowed inside a field of a debug command.

The only exception to this rule occurs when a character variable is specified in the *v* subfield of the first field of the SET command. In this case, blanks may appear inside quotes in the character constant in the *c* field.

This rule applies when a command contains a *p* subfield. This subfield, if specified in a command, is part of the first field of the command. Therefore, no blanks should appear before or after the slash (/) which separates the *p* subfield from the previous subfield, or before or after the colon (:) separator in the *p* subfield.

- Any number of blank characters (including zero) may appear between fields.

The only command with more than one field is the SET command, which has three. All other commands have either one field or none.

- The *v* (variable name) subfield in the DUMP, SET, and SETBP commands must be one of the following:
 - scalar variable name (including a function subprogram entry point name)
 - array element name (with constant subscripts)
 - array name (DUMP command only)

Note that a scalar or array subprogram parameter may be specified using one of these forms.

The variable *v* must appear in an executable statement in the designated program unit *p*, unless *p* is the main program or a block data program. If the COMPILER statement option DATA=AUTO or DATA=REUSE appeared in the program, then *v* must be a variable appearing in a COMMON block or a SAVE statement.

- The *p* (program unit) field in the PROG command and the *p* subfield in the DUMP, SET, SETBP, BREAK, CLEAR, and GO commands has the following format:

progrname[:*extname*]

where:

progrname represents the desired program unit in the ASCII FORTRAN program. It may be specified as (1) * (to represent the main program), (2) a FORTRAN program unit (main program, subroutine, function, or block data program) name, or (3) an unsigned positive integer *n* (to represent the *n*th unnamed block data program in the FORTRAN source program).

extname represents the program unit name of the external program unit corresponding to the internal subprogram *progrname*. Therefore, *extname* may be specified only if *progrname* is specified as a subprogram name which represents a FORTRAN internal subprogram. The variable *extname* may be specified as: (1) asterisk (*) (if the external program unit is the main program); or (2) a FORTRAN program unit (main program, subroutine, or function) name.

If *progrname* is specified as a program unit name and *extname* is not specified, then the external program unit with name *progrname* is taken. If no such external program unit exists, then the first internal subprogram with name *progrname* is taken.

Examples of the *p* format:

*	main program
3	third unnamed block data program in the source
SUB1	external program unit SUB1 (or, if no external program unit SUB1 exists in the program, the first internal subprogram in the source with name SUB1)
INT1:SUB2	internal subprogram INT1, whose external program unit is SUB2
INT2:*	internal subprogram INT2, whose external program unit is the main program

The checkout debug commands are described individually in the following subsections.

10.6.3.1. BREAK

Purpose:

The BREAK command (abbreviated B) sets break points at statement labels or source line numbers.

Form:

```
BREAK n [L [/ p ]] [, n [L [/ p ]]] ...
```

where:

n is an unsigned positive integer.

p represents a program unit in the FORTRAN source program. The *p* subfield is described under syntax rules (see 10.6.3).

Description:

The BREAK command specifies labels or a line number as points at which execution of the FORTRAN program is to be interrupted and interactive debug mode entered. These points are referred to as break points.

If the *n* L [/ *p*] format is used, then the break point is the beginning of the FORTRAN statement with statement label *n*, where *p* designates the program unit in which *n* resides. If *p* is not specified, then the break point is label *n* in the default program unit. (See PROG command, 10.6.3.10.)

If the *n* format is used, then the break point is the beginning of the FORTRAN statement at source line number *n*.

A maximum of eight label breaks and eight line number breaks may be set at any one time.

Two other debug commands are used in connection with the BREAK command. The CLEAR command is used to clear one or more break points. The LIST command is used to list all break points.

Example:

```
BREAK 10L/SUB1, 9
```

Set break points at (1) statement label 10 in program unit SUB1, and (2) source line 9. Debug mode will be reentered at these points. The command GO should follow so that the program resumes execution.

10.6.3.2. CALL

Purpose:

The CALL command (abbreviated CA) calls a FORTRAN subprogram.

Form:

CALL *s* [(*a*,*a* . . .)]

where:

s is a subprogram entry point name.

a is an actual argument that is passed to the subprogram.

Description:

The CALL command calls a FORTRAN subprogram with the given arguments. This allows the user to test only a given subprogram without having to execute the entire FORTRAN program. For example, a subprogram could be repeatedly called with different sets of arguments.

s has the following format:

ent[:*extname*]

where:

ent is the entry point to be called; *ent* may be any entry point in any subprogram in the FORTRAN program, except for an alternate entry point (that is, an entry point specified in an ENTRY statement) in an internal subprogram.

extname represents the program unit name of the external program unit corresponding to the internal subprogram *ent*. Therefore, *extname* may be specified only if *ent* is specified as an internal subprogram name. The variable *extname* may be specified as: (1) asterisk (*) (if the external program unit is the main program); or (2) a FORTRAN program unit (main program, subroutine, or function) name.

If *extname* is not specified, then the external subprogram entry point with name *ent* is taken. If no such external subprogram entry point exists, then the first internal subprogram with name *ent* is taken.

Each *a* is an actual argument and must match the corresponding formal argument of *s* in type and usage. (In this way, the CALL command closely resembles a subprogram reference in a FORTRAN program.)

a must be specified in one of the following forms:

- a FORTRAN constant.
- a variable in program unit *p*, where *p* is the default program unit (set by the PROG command). It must be specified as either a scalar variable name, an array name, or an array element name (with constant subscripts).

- a subprogram entry point name, immediately preceded by an asterisk (*). The name *a* may be any entry point in any subprogram in the FORTRAN program, except for an alternate entry point (that is, an entry point specified in an ENTRY statement) in an internal subprogram. If the entry point specified exists as an external subprogram entry point, then that one is taken. If no such external entry point exists, then the first internal subprogram with the specified name is taken.

Note that a statement label may not be passed as an actual argument via the CALL command. Therefore, a subprogram with any RETURN *i* statements (that is, a subprogram with * as any formal argument) may not be called with this command.

A maximum of 20 arguments is allowed.

When the subprogram returns (via the RETURN statement), control is transferred back to interactive debug mode. The message:

```
ENTER DEBUG MODE (RETURN FROM CALL COMMAND)
```

is printed. In addition, if the subprogram called was a function, the message:

```
FUNCTION VALUE RETURNED:
```

is printed, followed by the actual value.

Examples:

```
CALL FUNC1( A(1,1), 1.6E4, V)
```

Refer to function FUNC1, passing as arguments array element A(1,1) (from the default program unit), the real constant 1.6E4, and variable V (also from the default program unit). After FUNC1 returns control, the function value will be printed, and debug mode will be reentered.

```
CALL SUB1( 5, *SUB2)
```

Call subroutine SUB1, passing as arguments the integer constant 5 and subprogram entry point SUB2. After SUB1 returns control, debug mode will be reentered.

When debug mode is reentered on return from the CALL command, the user may not resume normal execution of the program (at the line where the CALL command was executed) using the GO command. Instead, the SAVE and RESTORE commands must be used for this purpose, since the CALL command interrupts normal execution.

For example, if the user wishes to execute a portion of the program, interrupt execution to test subprogram SUB (using the CALL command), and then resume normal execution of the program, the following commands could be entered:

```
SAVE  
CALL SUB  
RESTORE
```

10.6.3.3. CLEAR

Purpose:

The CLEAR command (abbreviated C) clears break points set by the BREAK and SETBP commands.

Form:
$$\text{CLEAR } \left\{ \left\{ \begin{array}{l} n \\ k \end{array} \right. [L[/p]] [, n [L[/p]]] \dots \right\}$$
where:

n is an unsigned positive integer.

p represents a program unit in the FORTRAN source program. The p subfield is described under syntax rules (see 10.6.3).

k is one of the following keywords: LABEL, LINE, BRKPT, or ALL.

Description:

The CLEAR command clears one or more break points established by the BREAK or SETBP commands.

The $n [L[/p]]$ field has the same format as the field in the BREAK command. The format $n L[/p]$ clears the break point set at statement label n in program unit p . The format n clears the break point at line n .

The rest of the formats are used to clear one or both break lists or the SETBP break. CLEAR LABEL clears all label break points. CLEAR LINE clears all line number break points. CLEAR BRKPT clears the breakpoint register set by the SETBP command. CLEAR and CLEAR ALL clear all label and line number breaks and the SETBP break.

LABEL, LINE, BRKPT, and ALL may be abbreviated to LA, LI, B, and A, respectively.

Example:

```
CLEAR 10L/SUB1, 9
```

Clear the break points (previously set by the BREAK command) at (1) statement label 10 in program unit SUB1, and (2) source line 9.

10.6.3.4. DUMP

Purpose:

The DUMP command (abbreviated D) prints the values of FORTRAN variables.

Form:

$$\text{DUMP } [, \text{opt}] \left\{ \begin{array}{l} v [/p] [, v [/p]] \dots \\ /p \\ ! \end{array} \right\}$$
where:

opt is an option letter; A or O is allowed.

v is a variable in program unit *p*. The *v* subfield is described under syntax rules (see 10.6.3).

p represents a program unit in the FORTRAN source program. The *p* subfield is described under syntax rules (see 10.6.3).

Description:

The DUMP command prints the current values of one or more FORTRAN variables.

If the O option is specified on the DUMP command, the values are printed in octal format. If the A option is specified, they are printed in ASCII character format. If neither O nor A is specified, they are printed in a format corresponding to the variable's data type (INTEGER, REAL, COMPLEX, etc.).

Whenever the value of a variable is printed, it is preceded by a heading line in the format:

$$v \quad /p$$

where *v* is the variable name and *p* is the program unit name. (See the description of the *p* subfield in 10.6.3.)

If an entire array is dumped, the values of all elements in the array are printed in column-major order.

The formats are described as follows:

■ DUMP [,opt] *v*[/*p*] [, *v* [/*p*]] ...

The *v*[/*p*] format prints the value of variable *v* in program unit *p*.

If *v* is a scalar, array element, or function entry point, then one value is printed. If *v* is an array name, then the values of all elements in the array are printed.

If *p* is not specified, then variable *v* is taken from the default program unit (set by the PROG command).

■ `DUMP [,opt] /p`

This format prints the values of all variables in program unit *p*.

■ `DUMP [,opt] !`

This format prints the values of all variables in all program units in the FORTRAN program.

If the second or third formats are used, then the order that the variables appear in the output is as follows:

- In a program unit, the variables are listed in alphabetical order.
- In the FORTRAN program (format 3), the program units are listed in the order that they appear in the source input.

Example:

`DUMP !`

Print the values of all variables in all program units.

`DUMP /SUB2`

Print the values of all variables in program unit SUB2.

`DUMP ,0 A/*, B(1,1), C/SUB3`

Print the values of variables A (from the main program), B(1,1) (from the default program unit), and C (from program unit SUB3), all in octal format.

10.6.3.5. EXIT

Purpose:

The EXIT command (abbreviated E) terminates the ASCII FORTRAN processor.

Form:

EXIT

Description:

The EXIT command terminates the processor with a call to the FEXIT\$ system routine. This routine terminates all input/output and does an ER EXIT\$.

Note that a "@" image (control statement) read in checkout debug mode is treated like an EXIT command (that is, the processor is terminated).

10.6.3.6. GO

Purpose:

The GO command (abbreviated G) resumes execution of an ASCII FORTRAN program.

Form:

GO [*n*L[/*p*]]

where:

n is an unsigned positive integer.

p represents a program unit in the FORTRAN source program. The *p* subfield is described under syntax rules (see 10.6.3).

Description:

The GO command causes an exit from interactive debug mode; execution of the FORTRAN program is then resumed.

If "GO" is specified (that is, no command fields), then execution of the program continues at the point at which it was interrupted to go into debug mode.

If the *n*L[/*p*] field is specified, execution of the program continues at statement label *n* in program unit *p*. If *p* is not specified, the default program unit (set by the PROG command) is assumed.

The user should be cautious when specifying the *n*L[/*p*] format, since registers may not be set up correctly when jumping to a statement label. For instance, jumping to a label inside a DO-loop or jumping to a label in another program unit (that is, not the one currently being executed) could cause execution problems.

Examples:

GO

Resume execution of the FORTRAN program at the point at which it was interrupted to go into debug mode.

GO 15L/S1

Resume execution of the program at statement label 15 in program unit S1.

10.6.3.7. HELP

Purpose:

The HELP command (abbreviated H) prints information about debug commands.

Form:
$$\text{HELP } \left[[,opt] \left\{ \begin{array}{l} \text{ALL} \\ cmd \end{array} \right\} \right]$$
where:

opt is an option letter; F or D is allowed.

cmd is one of the checkout debug command names. No abbreviations are allowed.

Description:

The HELP command prints information about debug commands, thereby allowing the user to continue debugging without having to consult a manual about command descriptions or formats.

The format HELP lists all of the debug command names.

The format HELP *cmd* prints all available information about the designated debug command *cmd*, including a list of all formats, a description of the individual items specified in the formats, and a general description of the command.

The format HELP ALL lists all available information for all debug commands. Note that a large amount of output is generated.

If the F option is specified, then only command formats will be printed.

If the D option is specified, then only command descriptions will be printed.

Examples:

HELP
List all of the debug command names.

HELP ALL
List all information for all debug commands.

HELP , F DUMP
List all formats for the DUMP command.

10.6.3.8. LINE

Purpose:

The LINE command (abbreviated LIN) prints the current source line number.

Form:

LINE

Description:

The LINE command prints the source line number of the statement in the FORTRAN program where execution was interrupted to go into debug mode.

10.6.3.9. LIST

Purpose:

The LIST command (abbreviated L) lists all break points set by the BREAK command and the default program unit.

Form:

LIST

Description:

The LIST command lists all break points set by the BREAK command. This includes all line number breaks and all statement label breaks.

The default program unit (set by the PROG command) is also listed.

10.6.3.10. PROG

Purpose:

The PROG command (abbreviated P) sets the default program unit for variables and statement labels.

Form:

PROG *p*

where *p* represents a program unit in the FORTRAN source program. The *p* field is described under syntax rules (see 10.6.3).

Description:

The PROG command sets the default program unit in the FORTRAN symbolic element that is implied for variables (in the DUMP, SET, SETBP, and CALL commands) and statement labels (in the BREAK, CLEAR, and GO commands) to *p*.

If no PROG command has been entered in debug mode during execution of the FORTRAN program, then the first program unit in the FORTRAN symbolic element is set as the default.

The default program unit set by this command may be overridden in an individual command (DUMP, SET, SETBP, GO, BREAK, or CLEAR) by specifying a program unit subfield *p* in that command.

The LIST command will print the default program unit.

Examples:

Assume the following commands are entered sequentially.

```
PROG SUB1
    Set program unit SUB1 as the default program unit.
DUMP X
    Print the value of variable X in the program unit SUB1.
DUMP X/2
    Print the value of variable X in the element's second unnamed block data program.
BREAK 10L
    Set a break at statement label 10 in program unit SUB1.
BREAK 10L/*
    Set a break at statement label 10 in the main program.
```


10.6.3.11. RESTORE

Purpose:

The RESTORE command (abbreviated R) restores the user's program to a previous point of execution.

Form:

RESTORE [*n*]

where *n* is an integer consisting of 1 to 12 digits.

Description:

The RESTORE command restores the state of the user's program which a previous corresponding SAVE command preserved (see 10.6.3.12), essentially restarting his program at the state it was in at the SAVE point. The optional version number *n* can be used to keep several stages of execution around when debugging.

When reentering the user program, the following message is printed out:

```
ENTERING USER PROGRAM: prog-name [ VERSION: version-no ]
```

NOTE: *The user is responsible for the assignment of files and their positioning. File contents, assignments, and positioning (tapes) are not saved or restored. Only the user's variables and point of execution are saved and restored, along with several debug mode parameters: break points set by the BREAK and STEP commands, the default program unit set by the PROG command, and the trace mode value set by the TRACE command. Also, the same level of ASCII FORTRAN must have been used to do the corresponding SAVE command.*

The user can reenter a checkout debugging run at a later date by use of the special Restart processor (FTNR) released with ASCII FORTRAN. All that is necessary is that the relocatable output file from the previous session still be available. (This is where the SAVE information is saved; see 10.6.6.)

A RESTORE command can also be done by use of the run-time routine CHKR\$ (see 7.3.3.10).

Examples:

```
@FTN,SCZ  IN.ELT  
@EOF
```

State is automatically saved in omnibus element IN.ELT before entry to debug mode.

```
BREAK 7  
GO
```

ASCII FORTRAN responds with BREAK AT LINE 7

```
SAVE 2
```

State saved in omnibus element IN.ELT/2

```
RESTORE
```

State restored from omnibus element IN.ELT.

ASCII FORTRAN responds with ENTERING USER PROGRAM: ELT

```
BREAK 3  
GO
```

User program now restarts execution from the original save point.
ASCII FORTRAN responds with BREAK AT LINE 3.

RESTORE 2

State restored from omnibus element IN.ELT/2.

ASCII FORTRAN responds with ENTERING USER PROGRAM: ELT VERSION: 2.

GO

User program resumes execution at line number 7, where the save was done.

@FTN,NCZ IN.GAMES,SAVE.GAMES

@EOF

The state is automatically saved in SAVE.GAMES before entry to debug mode.

GO

User program executes

@FIN

Next day the user wants to do more testing on GAMES.

@RUN SMITH,123456,TRNG

@FTNR SAVE.GAMES

FTNR responds with a sign-on line and the state of GAMES is restored from yesterday's SAVE.

GO

User GAMES program now executes again.

10.6.3.12. SAVE

Purpose:

The SAVE command (abbreviated SA) saves the present state of the user's program for later resumption.

Form:

SAVE [*n*]

where *n* is an integer consisting of 1 to 12 digits.

Description:

The SAVE command saves the present state of the user's program by writing it out to an omnibus element in a Relocatable Output (RO) file. The element name used is the RO element name. The version name used is either the user's RO version, if any, or the up to 12-digit field on the SAVE command.

Only an all-digit field may be used on the SAVE command. Since the element created is typed as omnibus, this command does not destroy the user's symbolic, relocatable, or absolute elements of the same name in his RO file.

The RESTORE (see 10.6.3.11) command may be used to restore the FORTRAN program to the state of execution of a corresponding SAVE command.

An automatic SAVE command is done for the user just before initially entering debug mode (after the END FTN message).

A SAVE command can also be done by use of the run-time routine CHKSV\$ (see 7.3.3.10).

Examples:

```
@FTN,SCZ  IN.ELT
@EOF
SAVE
```

This will save state in omnibus element IN.ELT . The RESTORE command can be used to restore the current state.

```
@FTN,NCZ  IN.ELT,OUT.ELT/TEST
@EOF
SAVE
```

This will save state in omnibus element OUT.ELT/TEST . The RESTORE command can be used to restore the current state.

```
@FTN,NCZ  IN.ELT,OUT.ELT/TEST
@EOF
SAVE 99
```

This will save state in omnibus element OUT.ELT/99 . The RESTORE command 99 can be used to restore the current state.

10.6.3.13. SET

Purpose:

The SET command (abbreviated S) changes the value of a FORTRAN variable.

Form:

SET v [$/p$] = c

where:

- v is a variable in program unit p . The v subfield is described under syntax rules (see 10.6.3).
- p represents a program unit in the FORTRAN source program. The p subfield is described under syntax rules (see 10.6.3).
- c is a FORTRAN constant.

Description:

The SET command sets the value of variable v in program unit p to the constant c . If p is not specified, then v is from the default program unit (set by the PROG command).

The variable c must be the same data type as v . There are no conversions between data types for the SET command. For example, if v is declared as type COMPLEX*16 in program p , then c must be a double precision complex constant.

If v is a character variable, then c must be a character constant. Hollerith constants are not allowed.

Examples:

SET I=5

Set the value of variable I (in the default program unit) to 5. I must be type integer.

SET CD(2,3)/S2 = (1.4D2, 2.3D3)

Set the value of array element CD(2,3) in program unit S2 to the COMPLEX*16 constant (1.4D2,2.3D3). Array CD must be type COMPLEX*16.

10.6.3.14. SETBP

Purpose:

The SETBP command (abbreviated SETB) sets a break point so that debug mode is reentered when a specific variable is set or referred to.

Form:

SETBP [*opt*] *v* [/*p*]

where:

opt is an option letter; R or W is allowed.

v is a variable in program unit *p*. The *v* subfield is described under syntax rules (see 10.6.3).

p represents a program unit in the FORTRAN source program. The *p* subfield is described under syntax rules (see 10.6.3).

Description:

The SETBP command sets a break point so that debug mode is reentered whenever the designated FORTRAN variable, *v*, in program unit, *p*, is set or referenced during execution of a FORTRAN program. If *p* is not specified, *v* is taken from the default program unit (set by the PROG command).

The SETBP command may be used only if execution is on a machine where the ER SETBP\$ mechanism is available. This Executive Request sets the programmable breakpoint register, which causes a breakpoint interrupt whenever the specified condition is met. Checkout debug mode is reentered at the beginning of the next executable FORTRAN statement after the specified variable has been set or referenced.

If the R option is specified on the SETBP command, then debug mode is reentered whenever the designated variable *v* is referenced from storage. This occurs when the variable is referenced in an assignment statement (on the right side of the assignment "=" operator) or an I/O write statement.

If the W option is specified, then debug mode is reentered whenever the variable *v* is stored into. This occurs when the variable is set in an assignment statement (on the left side of the assignment "=" operator) or an I/O read statement.

If neither the R nor the W option is specified, then both are assumed; that is, debug mode is reentered whenever the variable is set or referenced.

Note that a breakpoint interrupt will occur whenever the storage that the variable occupies is involved in a load (R option) or store (W option) instruction. Therefore, the FORTRAN statement where the breakpoint interrupt occurs (that is, the executable statement immediately preceding the statement where debug mode is reentered) may not actually reference the variable name specified in the SETBP command; the interrupt may have been caused by the setting or referencing of a variable that occupies the same storage as the variable designated in the command. Variables which may be overlapped in storage in a FORTRAN program include those used in EQUIVALENCE or COMMON statements or those passed as subprogram parameters.

The break point set by the SETBP command will remain in effect during execution until it is cleared by the CLEAR command (either CLEAR or CLEAR BRKPT format), or until another break point is set with the SETBP command. Note that only one SETBP break point may be set at any one time.

Examples:

SETBP,W C(6)/*

Set a break point so that debug mode is reentered whenever array element C(6) in the main program is set. The command GO should follow so that the program resumes execution.

SETBP,R V

Set a break point so that debug mode is reentered whenever variable V (in the default program unit) is referred to.

SETBP A/S2

Set a break point so that debug mode is reentered whenever variable A in program unit S2 is set or referred to.

10.6.3.15. SNAP**Purpose:**

The SNAP command (abbreviated SN) produces a dump of the FORTRAN program or registers.

Form:

SNAP [/]

where / is one of the following letters: I, D, or R.

Description:

The SNAP command dumps all or part of the FORTRAN program. The Executive Request ER SNAP\$ is used to dump the contents of one location counter at a time. The user program's registers may also be dumped.

The SNAP format dumps the entire FORTRAN program and all registers.

The SNAP I format dumps the contents of location counter 1 of the program. \$(1) contains all program instructions not resulting from input/output lists.

The SNAP D format dumps the contents of all location counters except 1.

The SNAP R format causes all registers to be dumped.

Since ER SNAP\$ lists absolute addresses only, an L option checkout compiler listing of the FORTRAN program may be helpful if the user wishes to decode the location counter information that is dumped. This listing includes the location counters and relative addresses for variables and code in the program.

10.6.3.16. STEP

Purpose:

The STEP command (abbreviated ST) sets a break point at a certain point ahead in the program.

Form:

STEP [*n*]

where *n* is an unsigned positive integer.

Description:

The STEP command specifies a break point at which execution of the FORTRAN program is to be interrupted and interactive debug mode reentered.

After the GO command is entered, *n* FORTRAN statements are executed and then debug mode is reentered.

If *n* is omitted, one is assumed.

Example:

STEP 3

Set a break point so that debug mode will be reentered after three FORTRAN statements have been executed. The command GO should follow so that the program resumes execution.

10.6.3.17. TRACE

Purpose:

The TRACE command (abbreviated T) turns trace mode on or off.

Form:

TRACE [*m*]

where *m* is one of the following keywords: ON or OFF.

Description:

Either TRACE or TRACE ON turns trace mode on. TRACE OFF turns trace mode off.

If trace mode is on, then a message in the form:

LINE *n*

is printed at the start of execution of each FORTRAN statement, where *n* is the source line number of the statement. Trace mode is initially off.

10.6.3.18. WALKBACK

Purpose:

The WALKBACK command (abbreviated W) traces the general flow of program execution through FORTRAN subprograms.

Form:

WALKBACK

Description:

The WALKBACK command gives a step-by-step trace of FORTRAN subprogram references that have occurred during program execution. The trace begins at the current statement in the subprogram which is executing (that is, the point in the user program at which execution was interrupted to go into debug mode) and ends at the main program.

During execution of the walkback trace, one line is printed at each step indicating which FORTRAN subprogram (subroutine or function) was referenced at a certain line number of another subprogram (or the main program). Walkback may occur over any number of subprograms.

Any FORTRAN subprogram named in a walkback message will be a main entry point, regardless of which entry point in that subprogram was actually referenced.

One or more of the following messages may be printed during the checkout walkback process:

WALKBACK INITIATED AT ADDRESS *absadr* IN USER PROGRAM

THIS ADDRESS IS AT LN. *line* OF *prog2*

prog1 REFERENCED AT LN. *line* OF *prog2*

THIS ADDRESS IS AT *elt \$(lc) reladr*

prog1 REFERENCED AT *elt \$(lc) reladr*

prog1 REFERENCED AT ADDRESS *absadr* BDI *bdi*

THIS ADDRESS IS IN THE I/O COMPLEX

X11 LINK ADDRESS DESTROYED - WALKBACK TERMINATED

WALKBACK TERMINATED BECAUSE OF AUTO. STORAGE

prog1 REFERENCED BY CALL COMMAND

Example:

```
1.  I = 5
2.  CALL S(1)
3.  END

4.  SUBROUTINE S(I1)
5.  J = F(I1)
6.  PRINT *,J
7.  RETURN
8.  END

9.  FUNCTION F(I2)
10. F = I2**3
11. RETURN
12. END
```

Execution of the following three checkout debug commands:

```
BREAK 11
GO
WALKBACK
```

causes the following lines to be printed:

```
WALKBACK INITIATED AT ADDRESS 031470 IN USER PROGRAM
THIS ADDRESS IS AT LN.    11 OF F
F    REFERENCED AT LN.    5 OF S
S    REFERENCED AT LN.    2 OF MAIN PROGRAM
```

See 10.7.4 for more walkback examples and a more complete description of the walkback mechanism. That subsection describes FTNWB, the run-time walkback routine, which is very similar to the checkout walkback process.

10.6.4. Contingencies in Checkout Mode

When an illegal operation (IOPR), guard mode (IGDM), or error mode (EMODE, including a math or I/O library error) contingency interrupt occurs during checkout execution of the user program, the following action will be taken (after a contingency message is printed), depending on the options specified on the @FTN control statement:

- C option (but no F or Z option)

A walkback trace will be performed (from the point of the error back to the main program). See 10.6.3.18 for a description of the walkback mechanism.

- CF options (but no Z option)

(1) A walkback trace will be performed, and (2) the current values of all variables in the user program will be printed (that is, the automatic 'DUMP !' command).

- CZ options

(1) A walkback trace will be performed, and (2) interactive debug mode will be entered (so that the user may attempt to find the problem). Execution of the user program may not be resumed after a contingency of this type.

When a break keyin (@@X C) occurs during checkout execution of the user program, the following action will be taken (depending on the @FTN options):

- C option (but no Z option)

No action will be taken.

- CZ options

(1) The message "BREAK KEYIN" will be printed; (2) the current FORTRAN statement will complete execution (so that debug mode is not entered in the middle of a statement); and (3) debug mode will be entered. Execution of the user program may be resumed with the GO checkout command.

When a divide fault (IDOF), floating-point overflow (IFOF), or floating-point underflow (IFUF) contingency interrupt occurs during checkout execution of the user program, the following will be performed (if the appropriate run-time counter is positive): (1) a contingency message will be printed, (2) the line number and program unit where the contingency occurred will be listed, and (3) execution of the user program will be resumed. See subsections on service routines DIVSET (7.3.3.9), OVFSSET (7.3.3.8), and UNDSSET (7.3.3.7) for a description of how the run-time counters are set. Note that all three counters are initialized to 20 on CZ options, and to zero otherwise.

10.6.5. Checkout Mode Restrictions

Due to the generation of code in main storage, only simple program structure can be provided in checkout mode. Links cannot be generated to subprograms that are not physically in the source program. In addition, multibanking and segmentation are not provided (that is, the BANK statement and the COMPILER statement with options BANKED=DUMARG, BANKED=ACTARG, BANKED=RETURN, or BANKED=ALL present cannot be used). The maximum size for a user program in checkout mode is somewhat smaller than the maximum size for a user program in noncheckout mode, due to the addressing space used by checkout execution-time requirements. A warning message will be printed if this maximum size is exceeded.

The error diagnostics associated with the checkout mode are given in Appendix I.

10.6.6. Restart Processor (FTNR)

FTNR is a small, separate processor, released with ASCII FORTRAN, which restarts previous checkout debugging sessions. It is called with the SI field holding a file-element name that is an omnibus-type element holding one of the user's runs saved from a previous session. (See SAVE command, 10.6.3.12.) The processor signs on in a standard manner and goes interactive after reloading the program.

Example:

```
▷@FTNR SAVEFILE.MYPROG
ENTERING USER PROGRAM: MYPROG
```

If a version name is given on the FTNR call statement, this element/version is the base level which is restored by a parameterless RESTORE command.

Example:

```
▷@FTNR F.GAME/A
sign-on line
ENTERING USER PROGRAM: GAME VERSION: A
▷RESTORE
```

The RESTORE command in this example restores GAME/A again. Note that FTNR can be used to reenter nonfull debug checkout runs if the CHKSV\$ run-time routine is used. See the CHKSV\$ and CHKRS\$ descriptions (7.3.3.11).

All checkout debugging commands are available in the FTNR (Restart) processor. The same level of FTN and FTNR must be used, or the restart will not work. This is true of the RESTORE command also (see 10.6.3.11); the same level of FTN and FTNR must have done the corresponding SAVE.

10.7. Walkback and the Interactive FTNPMD

10.7.1. Introduction

When a contingency interrupt occurs during execution of an ASCII FORTRAN program, the following debugging aids will automatically be executed:

- the ASCII FORTRAN walkback process (FTNWB)
- the ASCII FORTRAN interactive postmortem dump (FTNPMD)

The walkback mechanism gives a step-by-step trace of FORTRAN subprogram references that have occurred during program execution, from the point of the error condition back to the main program. Only subprograms in relocatable elements generated by ASCII FORTRAN compilations or in MASM elements using the walkback procedures (see 10.7.4.3) can be traced by the walkback process.

FTNPMD allows the user to interactively dump the current values of FORTRAN variables in the executing program. The variables which may be dumped are those which exist in elements which were generated with F-option ASCII FORTRAN compilations. If optimization is used, the values provided for the variables may not be the same as the value expected. This difference is the result of the elimination of unneeded stores. If the program is running in batch mode (see 10.7.5.1), FTNPMD will be executed only if the F option was specified on the @XQT control statement.

In addition to being called by the contingency routine, both FTNWB and FTNPMD may be initiated by calls from the user program.

If the walkback and interactive PMD debugging aids are to execute correctly, neither the Z nor R option should be specified on the @MAP control statement used for collection of the program. The @MAP,Z statement suppresses generation of diagnostic tables in the absolute element. The @MAP,R statement generates a Collector relocatable element; no @MAP,R relocatables should appear in the user program.

In addition, the Collector directive TYPE EXTDIAG should be used if any FORTRAN subprogram main entry points are unreferenced in the program. This statement will cause all entry points (referenced or not) to be inserted in the Collector's diagnostic tables.

If the program is multibanked, it must follow the specifications listed in Appendix H, since both FTNWB and FTNPMD make assumptions about the program's banking structure.

FTNPMD and FTNWB will not execute correctly if the name of a relocatable element produced by ASCII FORTRAN is changed with the @CHG or @COPY control statement.

10.7.2. Diagnostic Tables Generated by ASCII FORTRAN

If the F option is specified on the ASCII FORTRAN processor control statement (@FTN,F), the compiler produces a large set of diagnostic tables (also known as INFO-010 text) under location counter 3 of the generated relocatable element. These tables contain information about variables, arrays, statement labels, and line numbers in the program units of the FORTRAN element.

If the F option is not specified on the @FTN control statement, a smaller set of diagnostic tables (containing only the program unit information) is generated. These tables will only allow walkback through the program units in the element, with no line numbers appearing in the walkback messages.

When the relocatable element is mapped into an absolute element, the Collector handles the INFO-010 location counter (3) in a special manner. The contents of \$(3) are inserted in the absolute element, but are never part of the executing program. Therefore, the size of the diagnostic tables does not affect the size of the loaded program. FTNWB and FTNPMD read necessary INFO-010 tables from the absolute element into local buffers.

In order for the FTNPMD and FTNWB routines to get mapped into an absolute element, at least one @FTN,F relocatable must be mapped in.

10.7.3. Initiating FTNWB and FTNPMD

Both walkback and the interactive PMD are automatically executed (in that order) if any of the following contingency interrupts occur:

- error detected by the math library
- error detected by the I/O library (see G.8)
- illegal operation (IOPR)
- guard mode (IGDM)
- error mode (EMODE)

No registers are destroyed in FTNWB or FTNPMD. Therefore, when the registers are dumped by the ER EABT\$ (which is performed by the contingency routine after walkback and the interactive PMD have completed), the contents of all registers will be the same as when the error occurred.

The interactive PMD (but not walkback) is executed:

- When a break keyin (@@X C) contingency occurs during execution of the program
- When the program executes the FORTRAN statement CALL FTNPMD (the routine has no parameters)

Walkback (but not the interactive PMD) occurs when the program executes the FORTRAN statement CALL FTNWB (the routine has no parameters). In this case, the walkback process begins at the point of the call and traces back to the main program. Program execution then continues at the statement after the call.

If the program is executing in checkout mode (see 10.6), then neither FTNWB nor FTNPMD can be executed by any of the methods mentioned previously.

In order for FTNPMD to execute in batch mode in the above cases, an F option must be specified on the @XQT control statement. Note, however, that walkback does not require a special @XQT option for batch mode operation.

If the F option is specified on ASCII FORTRAN compilations (@FTN,F), normal line numbers will appear in the walkback messages; otherwise, all line numbers will be printed as zero (the subprograms' names will be correct, however).

10.7.4. Walkback (FTNWB)

10.7.4.1. Description of the Walkback Process

During execution of the walkback process, one line is printed at each step indicating which FORTRAN subprogram (subroutine or function) was referenced at a certain line number of another subprogram or the main program. Walkback may occur over any number of subprograms.

The walkback process will terminate when the trace has reached (1) the main program or (2) a routine which is in an element which was not generated by an ASCII FORTRAN compilation. The latter case includes ASCII FORTRAN service routines (see 7.3.3) and routines in MASM elements.

If the F option is specified on ASCII FORTRAN compilations (@FTN,F), normal line numbers will appear in the walkback messages; otherwise, all line numbers will be printed as zero (the subprograms' names will be correct, however).

In the examples listed in the following subsections, it is assumed that all ASCII FORTRAN compilations are performed with the F option, and that a type2 (banked I/O) library file is used as a LIB file during collection.

10.7.4.1.1. Errors Detected by the Math Library (CML)

The math library (common bank or relocatable) detects the following errors:

- unnormalized argument
- argument value out of range
- function value out of range

If any of these errors are detected during execution, messages will be printed, listing the following:

- the math routine which detected the error
- the nature of the error
- the decimal and octal representations of the argument which caused the error
- a walkback trace of the user subprogram references, from the call that referenced the math library, back to the main program (if possible). If the CMLSET service routine (see 7.3.3.10) has been called (and the CMLSET run-time counter is positive), a one-step walkback will be performed (listing the line number and program unit where math was called), and program execution will resume.

Example:

ASCII FORTRAN source input:

```
1.      CALL S1
2.      END

3.      SUBROUTINE S1
4.      CALL S2(-4.)
5.      RETURN
6.      END

7.      SUBROUTINE S2(X)
8.      Y=SQRT(X)
9.      RETURN
10.     END
```

Program execution:

```
ERROR TERMINATION IN SQRT ROUTINE CAUSED BY
ARGUMENT UNORMALIZED OR OUTSIDE ALLOWABLE RANGE
ARG1=          -4.0000000
ARG1 OCTAL 57437777777
SQRT REFERENCED AT ABSOLUTE ADDRESS 007740 BDI 000004
THIS ADDRESS IS AT LN.      8 OF S2
S2 REFERENCED AT LN.      4 OF S1
S1 REFERENCED AT LN.      1 OF MAIN PROGRAM
```

```
***** ENTER FTN PMD *****
```

```
-> (enter FTNPMD commands)
```

10.7.4.1.2. Errors Detected by the I/O Library

If a fatal error is detected during execution of any I/O library (common bank or relocatable) routine, then messages will be printed, listing the following:

- the nature and address of the error
- the I/O common bank or I/O relocatable element where the address resides
- a walkback trace of user subprogram references, from the call that referenced the I/O library, back to the main program (if possible)

Example:

ASCII FORTRAN source input:

```
1.      CALL S1
2.      END

3.      SUBROUTINE S1
4.      CALL S2
5.      RETURN
6.      END

7.      SUBROUTINE S2
8.      DIMENSION A(2)
9.      PRINT *,A(100000)
10.     RETURN
11.     END
```

Program execution:

```
GUARD MODE      ERR-CODE: 02
ERROR ADDRESS:  007603      BDI:  500025
THIS ADDRESS IS IN COMMON I/O BANK  C2F$
I/O      REFERENCED AT LN.   9  OF  S2
S2      REFERENCED AT LN.   4  OF  S1
S1      REFERENCED AT LN.   1  OF  MAIN PROGRAM
```

***** ENTER FTN PMD *****

-> (enter FTNPMD commands)

If a nonfatal I/O error is detected, then messages will be printed, listing the following:

- the nature of the error
- the line number and program unit where the error occurred

Program execution will then continue.

Example:

ASCII FORTRAN source input:

```
1.      CALL S1
2.      END

3.      SUBROUTINE S1
4.      READ (5,10) I
5. 10    FORMAT (I12)
6.      RETURN
7.      END
```

Program execution (assuming the character string 'A' is read by I/O):

```
FTN ERR ON UNIT-5      INPUT DATA DOES NOT CORRESPOND TO TYPE
I/O      REFERENCED AT LN.   4  OF  S1
```


10.7.4.1.3. Errors Detected in the User Program

If an error (illegal operation, guard mode, or error mode) is detected during execution of a user routine, then messages will be printed, listing the following:

- the nature and address of the error
- a walkback trace of user subprogram references, from the subprogram where the error occurred, back to the main program (if possible)

Example:

ASCII FORTRAN source input:

```
1.          CALL S1
2.          END

3.          SUBROUTINE S1
4.          CALL S2
5.          RETURN
6.          END

7.          SUBROUTINE S2
8.          DIMENSION A(2)
9.          A(100000) = 2.
10.         RETURN
11.         END
```

Program execution:

```
GUARD MODE      ERR-CODE: 02
ERROR ADDRESS:  003120      BDI: 000004
THIS ADDRESS IS AT LN.      9 OF S2
S2      REFERENCED AT LN.   4 OF S1
S1      REFERENCED AT LN.   1 OF MAIN PROGRAM
```

```
***** ENTER FTN PMD *****
```

```
-> (enter FTNPMD commands)
```

If a divide fault, floating-point overflow, or floating-point underflow contingency interrupt occurs during execution, and the appropriate run-time counter is positive, then messages will be printed, listing the following:

- the nature of the error
- the line number and program unit where the error occurred

Program execution will then resume.

See subsections on service routines DIVSET (7.3.3.9), OVFSSET (7.3.3.8), and UNDSSET (7.3.3.7) for a description of how the run-time counters are set.

Example:

ASCII FORTRAN source input:

```
1.      CALL S1
2.      END

3.      SUBROUTINE S1
4.      CALL DIVSET( 5 )
5.      A = 0.
6.      B = 1. / A
7.      RETURN
8.      END
```

Program execution:

```
WARNING: DIVIDE FAULT
AT LN.    6 OF S1
```

10.7.4.1.4. FTNWB Routine Call

If routine FTNWB is called during the execution of a user program, messages will be printed, listing the following:

- the address from which FTNWB was called
- a walkback trace of user subprogram references, from the subprogram which called FTNWB, back to the main program (if possible)

Example:

ASCII FORTRAN source input:

```
1.      CALL S1
2.      END

3.      SUBROUTINE S1
4.      CALL S2
5.      RETURN
6.      END

7.      SUBROUTINE S2
8.      CALL FTNWB
9.      RETURN
10.     END
```

Program execution:

```
FTNWB CALLED AT ADDRESS 003116  BDI 000004
THIS ADDRESS IS AT LN.    8 OF S2
S2      REFERENCED AT LN.    4 OF S1
S1      REFERENCED AT LN.    1 OF MAIN PROGRAM
```

10.7.4.2. Walkback Messages

One or more of the following messages may be printed during the walkback process:

FTNWB CALLED AT ADDRESS *absadr* BDI *bdi*

THIS ADDRESS IS AT LN. *line* OF *prog2*

prog1 REFERENCED AT LN. *line* OF *prog2*

THIS ADDRESS IS IN THE I/O COMPLEX

THIS ADDRESS IS AT *elt* \$(*lc*) *reladr*

prog1 REFERENCED AT *elt* \$(*lc*) *reladr*

prog1 REFERENCED AT ADDRESS *absadr* BDI *bdi*

X11 LINK ADDRESS DESTROYED – WALKBACK TERMINATED

AUTO. STORAGE ELEMENT ENCOUNTERED – WALKBACK TERMINATED

The values inserted in these messages are described as follows:

absadr absolute address (octal)

bdi index of the bank that *absadr* resides in (octal)

line source line number of a designated FORTRAN statement (decimal)

prog2 one of the following:

- MAIN PROGRAM (if the walkback process has reached the last step of a successful trace)
- an external subprogram name
- *i:e*, where *i* is an internal subprogram name and *e* represents *i*'s external program unit. The variable *e* may be either MAIN PROGRAM or an external subprogram name.

prog1 one of the following:

- I/O (to represent the I/O library complex)
- an external subprogram name
- *i:e* (see *prog2*)

elt an element name

lc a location counter in element *elt* (decimal)

reladr a relative address under location counter *lc* (octal)

Any FORTRAN subprogram named in a walkback message will be a main entry point, regardless of which entry point in that subprogram was actually referenced.

The last five walkback messages listed above are printed only when the trace has reached a termination point that is not in the main program.

10.7.4.3. Walkback Procedures for MASM Subprograms

There are two MASM procedures in the ASCII FORTRAN library, F\$EP and F\$INFO. These procedures can be referenced in a user's MASM element which is generating a subprogram to be used in an ASCII FORTRAN system. If these procedures are used correctly, ASCII FORTRAN's run-time walkback mechanism will properly handle the MASM subprogram.

10.7.4.3.1. F\$EP

The form of the F\$EP call line is as follows:

```
F$EP  sub
```

where *sub* is a field of six Fielddata characters (left-justified, blank-filled) denoting the subprogram name that is to be externalized by the procedure.

F\$EP generates some epilog and prolog code in the MASM subprogram, namely:

- Epilog: restore registers and return to the caller of the subprogram. The return is done via the IBJ\$ return mechanism (see 8.5.3); that is, a jump or LIJ is performed, depending on the contents of H1 of X11.
- The subprogram entry point *sub* (note that F\$EP externalizes the entry point, so *sub* should not appear as a label in the user's MASM code).
- Prolog: save registers (including X11).

The prolog-epilog code is the same as that generated by a @FTN,0 compilation (that is, over-65K D-bank code). Volatile registers A2 (prolog) and A4 (epilog) are destroyed in the generated code.

10.7.4.3.2. F\$INFO

The form of the F\$INFO call line is as follows:

```
F$INFO elt
```

where *elt* is a field of 12 Fielddata characters (left-justified, blank-filled) denoting the relocatable element name from the MASM control statement.

F\$INFO generates the following:

- the jump back up to the epilog code (which was generated by F\$EP)
- the INFO-010 diagnostic text (readable by the walkback routines in the ASCII FORTRAN library)

The INFO-010 tables are never loaded at run time along with the user program's I-bank and D-bank. Instead they are in the absolute element's diagnostic tables, which are read in by ASCII FORTRAN's run-time walkback routines. The tables contain information which allows these routines to determine the program unit and line number for walkback messages.

10.7.4.3.3. Description

If procedures F\$EP and F\$INFO are used, only one subprogram may appear in an element (that is, each of the two procedures should be referenced only once per MASM assembly).

A user's MASM element which references the two procedures and which contains the externalized subprogram SUB1 will appear as follows:

```
@MASM, IS  ELT1,ELT2
      AXR$
```

```
      F$EP   'SUB1
$(1)
```

```
      $(1) subprogram code except for epilog
      (restore registers and return to caller),
      prolog (save registers), and jump to epilog.
      Note that subprogram label (SUB1 in this
      case) is externalized by procedure F$EP.
      It should not appear as a label in the
      user's $(1) code.
```

```
      F$INFO 'ELT2
      END
```

The epilog and prolog code generated by these two procedures does not handle arguments (passed to the subprogram) or functions (that is, the MASM subprogram as a function). The user may process these items in the \$(1) subprogram code, if applicable. Arguments should be handled after the F\$EP reference. If the subprogram is a function, then A0 (and possibly A1, A2, and A3, depending on the function type) should be loaded with the function result before the F\$INFO call.

The user should not use \$(3) or \$(4) in a MASM element referencing procedures F\$EP and F\$INFO, as these location counters are used by the two procedures.

A MASM error (E-flag) will result if the user has an incorrect reference to one of the two procedures. Possible errors include:

- one of the procedures referred to more than once
- F\$INFO referred to without a previous reference to F\$EP
- \$(3) or \$(4) used in the user's code
- more than one field with one subfield passed to either procedure

- parameter passed to either procedure is not a Fielddata string
- the length of the Fielddata string passed to one of the procedures is not correct (six for F\$EP, 12 for F\$INFO)

Note that if the element containing the two procedures (ASCII FORTRAN library element INFO-PROC, marked as type ASMP) has a PDP performed on it by the user, an M option must appear on the PDP call statement.

Example:

```
@MASM, IS TEST
  AXR$
  F$EP      'SUB1
$(1)
  LX, U     AO, 0
  LXI, U    X11, 0
  LMJ      X11, SUB2
  F$INFO    'TEST
  END
```

```
@FTN, ISF MAIN

1.  CALL SUB1
2.  END

3.  SUBROUTINE SUB2
4.  CALL FTNWB
5.  END
```

If these two relocatable elements are collected into an absolute element, then the resulting program will execute as follows:

```
FTNWB CALLED AT ADDRESS 066431  BDI 000004
THIS ADDRESS IS AT LN.      4  OF  SUB2
SUB2  REFERENCED AT LN.     0  OF  SUB1
SUB1  REFERENCED AT LN.     1  OF  MAIN PROGRAM
```

Note that the line number in a MASM subprogram will always be zero in a walkback message.

10.7.5. Interactive Postmortem Dump (FTNPMD)

10.7.5.1. Soliciting Input

When execution is in interactive PMD mode, commands are solicited with ER TREAD\$. The solicitation message is "-".

If the command is read from an add stream (@ADD), or if the program began execution in breakpoint mode (@BRKPT), the command image will be printed.

The message:

```
***** ENTER FTN PMD *****
```

is printed on entry to interactive PMD mode.

To leave interactive PMD mode, the EXIT command (see 10.7.5.2.2) is used.

If the program is executing in batch mode, then PMD mode does not go interactive. Instead, if the F option was specified on the @XQT control statement, the two commands DUMP ! and EXIT are automatically executed. FTNPMD will not be executed in batch mode if the F option was not specified.

10.7.5.2. PMD Mode Commands

Both of the interactive PMD commands described below may be abbreviated to one letter (the initial one).

10.7.5.2.1. DUMP

Purpose:

The DUMP command (abbreviated D) prints the values of FORTRAN variables.

Form:

DUMP[,opt] { [v] / [p] / e }

where:

opt is an option letter; A or O is allowed.

v is the name of a FORTRAN variable in program unit *p* in element *e*. It must be one of the following:

- scalar variable name (including a function subprogram entry point name)
- array name
- array element name (with constant subscripts)

Note that a scalar or array subprogram argument may be specified using one of the these forms.

The variable v must appear in an executable statement in p , unless p is the main program or a block data program. If the ASCII FORTRAN compilation for e had a COMPILER statement option DATA=AUTO or DATA=REUSE, then v must be a variable in a COMMON block or a SAVE statement.

p represents a program unit in element e . It has the format:

progrname [: *extname*]

where:

progrname represents the desired program unit in the ASCII FORTRAN program. It may be specified as: (1) * (to represent the main program); (2) a FORTRAN program unit (main program, subroutine, function, or block data program) name; or (3) an unsigned positive integer n (to represent the n^{th} unnamed block data program in the FORTRAN source program).

extname represents the program unit name of the external program unit corresponding to the internal subprogram *progrname*. Therefore, *extname* may be specified only if *progrname* is specified as a subprogram name which represents a FORTRAN internal subprogram. The variable *extname* may be specified as: (1) * (if the external program unit is the main program); or (2) a FORTRAN program unit (main program, subroutine, or function) name.

If *progrname* is specified as a program unit name and *extname* is not specified, then the external program unit with name *progrname* is taken. If no such external program unit exists, then the first internal subprogram with name *progrname* is taken.

e is the name of a relocatable element that is mapped into the executing program and that was produced by an F option ASCII FORTRAN compilation.

General Description:

The DUMP command prints the current values of one or more FORTRAN variables.

If the O option is specified on the DUMP command, the values are printed in octal format. If the A option is specified, they are printed in ASCII character format. If neither O nor A is specified, they are printed in a format corresponding to the variable's data type (INTEGER, REAL, COMPLEX, etc.)

Whenever the value of a variable is printed, it is preceded by a heading line in the format:

v / p / e

where v is the variable name, p is the program unit name (see the description of the p subfield above), and e is the relocatable element name.

If the v subfield is specified and is a scalar, array element, or function entry point, then one value is printed. If v is specified and is an array, then all elements of the array are printed in column-major order.

The following syntax rules apply for the DUMP command:

- One or more blanks must appear between the command name (or command name with option) and the command's only field.
- No blank characters are allowed inside the command's only field. A blank character in the field terminates the syntax scan.

Form Descriptions:

1. DUMP [,opt] *v/p/e*

This format prints the value of variable *v* in program unit *p* in element *e*.

2. DUMP [,opt] *v//e*

This format prints the value of variable *v* in the first program unit in element *e*.

3. DUMP [,opt] */p/e*

This format prints the values of all variables in program unit *p* in element *e*.

4. DUMP [,opt] *//e*

This format prints the values of all variables in all program units in element *e*.

5. DUMP [,opt] *!*

This format prints the values of all variables in all program units in all relocatable elements produced by F option ASCII FORTRAN compilations that were mapped into the executing program. Before the variables in a given element *e* are dumped, a heading line is printed:

```
>>> ELEMENT e          <<<
```

If formats 3 through 5 are used, the variables appear in the dump in the following order:

- In a program unit, the variables are listed in alphabetical order.
- In an element, the program units are listed in the order that they appear in the FORTRAN element.
- If format 5 is used, the elements are listed in the order that their diagnostic text appears in the absolute element's diagnostic tables.

Formats 1 through 4 may be appended with two subfields after the *e* subfield. The format is:

```
DUMP [ ,opt] [v] / [p] / e [ / [s] / i]
```

where *s* and *i* represent the segment name and bank name, respectively, under which \$(3) of the relocatable element *e* is mapped. If *i* is specified and *s* is not, then the segment in bank *i* with segment index 0 is assumed.

The two extra subfields may be specified when the executing program is multibanked, although they are never required. If specified, they will cause the interactive PMD to search more efficiently through the absolute element's diagnostic tables for the diagnostic text corresponding to relocatable element *e*. The search is more efficient because the absolute element's pointer tables for the diagnostic text are organized by segment and bank, and not by element.

If the fourth and fifth subfields are not specified, FTNPMD will search sequentially through the absolute element's diagnostic tables when looking for the diagnostic text of relocatable element *e*. The diagnostic pointer tables in the absolute element are not used in this case.

Examples:

DUMP !

Print the values of all variables in all program units in all elements.

DUMP //ELT1

Print the values of all variables in all program units in element ELT1.

DUMP,0 /SUB1/E2

Print the values of all variables in subprogram SUB1 in element E2 (in octal format).

DUMP,A A//E3

Print the value of variable A in the first program unit of element E3 (in ASCII character format).

DUMP B(6,3,1)/*E4

Print the value of array element B(6,3,1) in the main program (in element E4).

10.7.5.2.2. EXIT

Purpose:

The EXIT command (abbreviated E) terminates the interactive PMD mode.

Form:

EXIT

Description:

The EXIT command causes program execution to leave the interactive PMD mode.

- If walkback and the PMD mode were initiated by a call from the contingency routine because of one of the five possible error conditions (math library error, I/O library error, illegal operation, guard mode, or error mode), control is returned to the contingency routine, where an ER EABT\$ is performed.
- If the PMD mode was initiated by a break keyin (@@X C) contingency, control is returned to the point of interrupt in the executing program.
- If the PMD mode was initiated by execution of the FORTRAN statement CALL FTNPMD, normal execution of the program is resumed.

The message:

***** EXIT FTN PMD *****

is printed when the interactive PMD mode is terminated.

Note that execution of the program is immediately terminated if a "@" image (control statement) is read in PMD mode. The FEXIT\$ termination routine is called.

10.7.5.3. FTNPMD Diagnostics

The diagnostics printed by the interactive PMD are explained in the following list.

BANK NOT FOUND

The bank specified in the *i* subfield of the DUMP command does not exist in the executing program, or no entry exists in the absolute element's diagnostic pointer tables corresponding to segment *s* (specified in the *s* subfield) in bank *i*.

ELEMENT MUST BE SPECIFIED

The *e* subfield in the DUMP command is required (unless the format DUMP [*opt*] ! is used).

ELEMENT NOT FOUND

The element specified in the *e* subfield of the DUMP command was not mapped into the executing program, or no diagnostic text was generated for the element.

ENTIRE ASSUMED-SIZE ARRAY CANNOT BE DUMPED

The range of an assumed-size array is not known. Only individual elements of an assumed-size array can be dumped.

FTEMP\$ STORAGE DESTROYED

A subprogram's temporary storage area (for saving registers and the argument list) has been destroyed because of an error in the user program. The specified variable cannot be dumped.

FUNCTION HAS NOT BEEN CALLED

An attempt has been made to dump a character function value (that is, the *v* subfield of the DUMP command was specified as a character function subprogram entry point), but the function has not yet been called during execution.

I MUST BE SPECIFIED

The *s* subfield (segment name) was specified in the DUMP command, but the *i* subfield (bank name) was not.

ILLEGAL COMMAND

An illegal PMD command name was specified when interactive PMD mode solicited input.

ILLEGAL SYNTAX

A general syntax error was found in the command image. This includes specifying a field for a command when none is allowed, or not specifying a field when one is required.

INCORRECT NUMBER OF SUBSCRIPTS

The number of subscripts specified for the array element in the ν subfield of the DUMP command does not equal the number of dimensions declared for the array in the specified program unit (p subfield) and element (e subfield).

**NO DIAGNOSTIC TABLES - WALKBACK & FTN PMD TERMINATED
USE F OPTION ON @FTN CARDS TO ENABLE THESE FEATURES**

In the collection process for the executing program, there were no elements with INFO-010 text mapped in (that is, no relocatable elements produced by ASCII FORTRAN compilations). Therefore, neither walkback nor the interactive PMD can execute, since they both require the use of the INFO-010 diagnostic tables.

PARAMETER'S SUBPROGRAM HAS NOT BEEN CALLED

An attempt has been made to dump a subroutine or function parameter (that is, the ν subfield of the DUMP command was specified as a subprogram parameter), but the subprogram has not yet been called during program execution.

PROGRAM UNIT NOT FOUND

The program unit specified in the p subfield of the DUMP command does not exist in the specified element (e subfield).

SEGMENT NOT FOUND

The segment specified in the s subfield of the DUMP command does not exist in any bank of the executing program.

SUBSCRIPT OUT OF RANGE FOR ARRAY

The constant subscripts specified for the array in subfield ν of the DUMP command is too large or too small.

VARIABLE IS NOT AN ARRAY

The variable in subfield ν of the DUMP command is appended with a subscript list, but the variable is not declared as an array in the specified program unit (p subfield) and element (e subfield).

VARIABLE NOT FOUND

The variable in subfield ν of the DUMP command does not exist in the specified program unit (p subfield) and element (e subfield), or e was not compiled with @FTN,F (thereby producing no diagnostic information about user variables).

WARNING: BAD DIAGNOSTIC TEXT FOUND; SEARCH TERMINATED

It is not possible to continue reading diagnostic (INFO-010) text from the absolute element's diagnostic tables, because bad text has been encountered. The bad text could be from (1) a

processor other than ASCII FORTRAN or FORTRAN V (which has generated a relocatable element with INFO-010 text) or (2) an old level of ASCII FORTRAN (which has generated a format of INFO-010 text that cannot be interpreted by the current FTNPMD and walkback run-time routines).

10.8. Compiler Optimization

The compiler always performs the following computational optimizations in addition to its other functions (see 1.3.1):

- The reduction of expressions involving constants to a single constant (for example, $3.14159**2$ is replaced by the single constant 9.8695877).
- The reordering of logical expressions to reduce the amount of time spent evaluating the expression (for example, `IF(A.OR.B) GO TO 10` is replaced by `IF(A) GO TO 10`, and `IF(B) GO TO 10`).
- All available registers are used to hold intermediate results of calculations and reduce the number of references to slower storage.

Using processor options, the compiler may be directed to devote additional time to optimizing the FORTRAN statements before generating the relocatable binary object program from the object code. This is done at the expense of compilation speed, but the resulting output will usually execute significantly faster than without this additional optimization.

10.8.1. Local Optimization

At the user's option, the compiler will partition the FORTRAN program into basic blocks, that is, a sequence of statements with no entry or exit points interior to the sequence. Within each of these blocks it will perform the following additional optimizations:

- The evaluation of most expressions which are constant at compilation time.
- The elimination of a computation, which at execution time will have already been computed in a previous statement, and replacement of the second computation by a reference to the temporary storage location or arithmetic register saving the result of the first computation. For example, the calculations of the following statement sequence would be done at compile time rather than at execution time:

```
D = 2
A = SQRT(D)
B = A
```

- The elimination of unneeded references to storage, within a basic block.
- The replacement of an operation with a faster, equivalent one (for example, $J*2$ is replaced by a shift operation).
- Simplification of more complex expressions. For example, the expression $A**2$ is replaced by $A*A$.

Local optimization is specified by the letter V in the options list of the @FTN control card.

10.8.2. Global Optimization

At the user's option, the compiler will perform an analysis of the interconnection patterns between basic blocks and perform the following additional optimizations:

- The elimination of redundant computations between basic blocks and the replacement of the second computation by a reference to the temporary storage location or arithmetic register which at execution time will hold the result of the first computation.
- The elimination of unneeded stores to variables across the whole program unit.
- Movement of computations which are constant relative to a loop to a point outside the loop, and the replacement of the computation within the loop by references to the temporary storage location or arithmetic register which will hold the result of the computation.
- Replacement of loop control variables, and expressions involving loop control variables by temporaries which are incremented on each pass through the loop.
- The maintenance of the result of computations and frequently used values in registers when execution crosses between blocks.

Global optimization is effected by specifying the letter Z in the options list of the @FTN control card. See code reordering in 8.5.4.

10.8.3. Optimization Pitfalls

The optimizer cannot fully optimize expressions involving local variables shared between an internal subprogram and its external program unit. Therefore, the programmer who wishes complete optimization of an element is cautioned to be careful of which data and how much data is shared by internal and external program units. The same situation exists with common blocks and arguments to subprograms. The optimizer operates only on one program unit at a time and can optimize operations on data local to the program unit most efficiently.

For example:

```
X=22.  
CALL Y(Z)  
X=X+1.
```

In this situation, if X is a local variable, the optimizer can keep the value in a register across the CALL to Y, since no other program unit can access purely local variables not passed as arguments.

If X were in common or shared between the external program unit and any internal subprograms, this optimization could not be done since the procedure Y could conceivably change the value of X.

10.9. Hints for Efficient Programming

The following list of suggestions should be followed to obtain the most efficient object programs from the ASCII FORTRAN compiler.

1. Simplify program structure as much as possible.

Efforts in debugging and maintaining a FORTRAN program will be greatly reduced and, in general, more optimal code will be produced by the compiler if unnecessary complexity is avoided. Individual statements should be simple enough to be easily understood. Loop nesting should be simple and straightforward with little inside branching. Program flow should be simple enough to minimize the number of paths through the program and make the purpose and method of the program easily understandable. Structured programming practices should be adhered to. Use the blocking statements (see 4.4) instead of GO TO statements whenever possible. Unstructured methods such as extended range of DO-loops should be avoided.

2. A number of limitations have been built into the optimizing portion of the compiler. The user may find it beneficial to break a program into additional subroutines if compiler messages indicate that the number of program variables or basic blocks which can be handled by optimization has been exceeded. When these limits are reached, the compiler will undertake a degraded form of optimization and continue the compilation, producing correct, although possibly less than optimal, code.
3. Although the compiler has been designed to handle variables in COMMON or EQUIVALENCE statements in the most optimal way, the compiler must assume the worst possible cases when references are made to user subroutines or functions. Avoid any unnecessary use of COMMON or EQUIVALENCE statements in order to aid optimization.
4. Because of the additional complexity involved in addressing variables with over 65K addresses, this alternative should only be used for programs which actually exceed 65K. This particularly applies to arrays in very tight loops.
5. Since references to dummy arguments (which are not arrays) require indirect references to storage, it is best to assign such variables to local variables for use within the subprogram, especially within loops.
6. Excessive nesting of loops may require the loading and unloading of registers, adding to the execution time of the program, and should be avoided.
7. Because of the way logical statements are optimized, the FORTRAN programmer should try to order the logical statements so that the most frequently satisfied comparison is performed first in a left to right sequence.
8. The initialization portion of a loop is assumed to be executed less frequently than the loop. When this assumption is violated, a FORTRAN program could take longer to execute with optimization than without.
9. The use of lists with ASSIGNED GO TO statements will help optimization by reducing the worst case assumptions that must be made without the lists. However, the list must include all labels which can be branched to at execution time or the compiler may produce incorrect code, and therefore, the execution of the program may produce incorrect results.
10. One portion of global optimization involves attempting to take advantage of the linear allocation of storage for multi-dimensional arrays. This involves the attempt to convert a data transfer operation within a simple loop into a single, linear operation referred to as a block transfer. A simple loop is defined as involving only the transfer of one array to another, or a constant into

an array. To take advantage of this form of optimization each array should be initialized or redefined within its own loop. Character arrays which start on word boundaries and whose element lengths are a multiple of 4 can also be optimized in this way.

10.10. Diagnostic System

The compiler contains a large number of self-explanatory diagnostic messages, some of which contain symbolic names from the source code. Diagnostic messages are printed adjacent to the source program statements that contain the errors or they are printed following the source statement listing if they pertain to the entire program unit.

Some messages are merely reminders to the programmer and do not affect the generated code. This type of message is prefaced with the word **WARNING** when it is printed.

Other messages indicate that more serious errors have been detected. A message prefaced by the word **ERROR** indicates that the code generated is possibly incorrect due to ambiguous or nonstandard usage of the source language. The programmer should check the generated code to ascertain whether it will correctly perform the operations intended. When an error is detected during a compilation, the relocatable binary element produced is marked as being in error. If the program is being run in batch mode, this will allow some execution where execution would normally be stopped, as in the case of detection of an I/O error.

Nonstandard usage messages occur only when the T option has been specified. See 10.5.1 (Processor Call Options).

The format of an error/warning/nonstandard usage message is:

* $\left. \begin{array}{l} \text{ERROR} \\ \text{WARNING} \\ \text{NON-STD USAGE} \end{array} \right\} x \text{ [at line } y] \text{ description}$

where x is a code number indicating the type of error or warning. See Appendix D for details on these codes. y indicates a source line number or a line number within an included procedure. If it is the latter, the line is given in the form:

proc.proc-line-number

where *proc* is the name of a procedure. (See 8.2.)

All diagnostic message descriptions are listed in Appendix D.

Regardless of the type of errors detected in the program, all statements are scanned to detect syntactical errors. At the end of the compilation listing (in the END FTN message), the numbers of warnings, errors, and nonstandard usages detected (if any) are printed.

Appendix A. Differences Between SPERRY UNIVAC FORTRAN Processors

A.1. General

The differences in language capabilities and equivalent syntax for ASCII FORTRAN and its predecessor, FORTRAN V, are discussed in this appendix.

A.2. Extensions to SPERRY UNIVAC FORTRAN V

1. An expanded character set (which handles exclusively ASCII data sets) is available.
2. Double precision complex data type (COMPLEX*16) is permitted.
3. A character data type is allowed.
4. The "\$" is allowed in symbolic names, except for the first character of a name.
5. General expressions for subscripts are allowed.
6. Concatenation of character strings is allowed.
7. Multiple assignments are implemented.
8. Character assignments and comparisons are permitted.
9. An integer expression for the index of a COMPUTED GO TO is permitted.
10. An optional comma in a DO statement is permitted.
11. Integer expressions for DO parameters are allowed.
12. The last statement of a DO range may be any statement which permits execution of the following statement.
13. The PAUSE statement is extended to allow longer messages. FORTRAN V STOP and PAUSE arguments should be enclosed in apostrophes to conform with ASCII FORTRAN.
14. The STOP statement is extended.

15. Expressions are permitted in an output I/O list.
16. Expressions are permitted in an implied-DO.
17. List-directed I/O statements are permitted.
18. ASCII FORTRAN enhances the use of storage greater than 65K words. ASCII FORTRAN programs which are larger than 65,535 words are made possible by the O option on the @FTN control command (see 6.9 and 10.5). Programs larger than 262,143 words are possible by also using the BANK statement (see 6.6 and 6.9). Accessing of arrays greater than 65K words was possible in FORTRAN V using the XM=1 compiler option.
19. Initialization of ASSIGN variables in a DATA statement is permitted.
20. The BANK statement permitting utilization of banks is implemented.
21. An extension to the EXTERNAL statement (&, *) is implemented.
22. An expanded set of mathematical library functions is allowed.
23. New service subroutines are provided.
24. The SUBSTR and BITS pseudo-functions are implemented.
25. A new DEBUG facility is implemented.
26. An extended form of the FORTRAN V PARAMETER statement is implemented.
27. A statement label variable may be used for the format number in an input/output statement.
28. Exponentiation between variables of all arithmetic types and lengths is permitted.
29. In data initialization statements, ASCII FORTRAN requires that the variables or array elements match the type of the corresponding constants if the type is real, complex, double precision, integer, or logical. When such a mismatch occurs, the compiler prints a diagnostic and converts the constant to match the type of the variable. FORTRAN V does not require element type to match the value type but does not convert the constant regardless of its type, possibly causing problems later in the FORTRAN program.
30. Checkout mode is available.
31. INCLUDE statements may specify a file name as well as an element name.
32. ASCII FORTRAN levels higher than 8R1 conform to FORTRAN 77 (ANSI X3.9-1978), and FORTRAN V conforms to FORTRAN 66 (ANSI X3.9-1966). Therefore, all features in FORTRAN 77 which were not in FORTRAN 66 are implemented in ASCII FORTRAN.

A.3. Exceptions to SPERRY UNIVAC FORTRAN V

1. Character data is represented in the ASCII code set rather than the Fieldata code set.

Although most Fieldata characters have corresponding representations in ASCII, characters are four to a word rather than six and will have different internal representations.

The three Fieldata graphic characters Δ , \square , and \neq have no equivalents in the ASCII graphic set. They are converted to caret (^), quotation mark ("), and underscore(_), respectively, on an ASCII printer.

2. ASCII FORTRAN does not allow transfer of control to labels on nonexecutable statements. However, FORMAT statements are allowed as targets of DO-loops.
3. The interpretation of RETURN k is a branch to the k^{th} statement label, and not to the k^{th} argument.

To obtain the effect of the FORTRAN V statement RETURN k , the programmer should:

- Create an explicit typing statement for an integer array ARR with as many elements as there are arguments in the subroutine argument list.
- Initialize the array such that when the k^{th} argument of the list is a statement label, the k^{th} element of the array is the sequential position of that statement label among all the statement labels in the list. Note that if there are ENTRY points in the subprogram, the array may need to be set separately in the flow of control after each ENTRY.
- Set all other array elements to zero.
- Change RETURN k to RETURN ARR (k).

For example, the subroutine:

```
SUBROUTINE SAM (Y, *, *, X)
.
.
.
      I=2
RETURN I
.
.
RETURN 3
END
```

should be changed to:

```
SUBROUTINE SAM (Y,*,*,X)
INTEGER IA (4)/0,1,2,0/
```

```
      I=2
RETURN IA(I)
```

```
      RETURN IA(3)
      END
```

4. The ABNORMAL statement is not implemented. (The compiler assumes that all FORTRAN mathematical functions are normal and all external procedures are abnormal. It will generate an error message and ignore all ABNORMAL statements.)
5. No FORTRAN V "hardware" IF statements are allowed (subroutine calls are available).
6. FORTRAN V permits a dummy argument to appear in an EQUIVALENCE statement, ASCII FORTRAN does not.
7. The FORTRAN V function INSTAT is not available in ASCII FORTRAN. It has been replaced by the functions IOC, IOS, and IOU (see 5.8.1).
8. Exponents have been standardized. ASCII FORTRAN exponents contain three digits rather than two digits used by FORTRAN V.
9. ASCII FORTRAN does not support any of the FORTRAN V options for the COMPILER statement. The ramifications of this and possible programmer actions are listed by option and variation in the following discussion:

■ FLD Option:

Although there is no counterpart for the FORTRAN V FLD option in ASCII FORTRAN, the FLD function is equivalent to the BITS pseudo-function. It may also be simulated via a statement function.

The effect of the FLD option is to alter the compiler interpretation of the FLD function. These alterations can be carried over to the BITS function as follows:

- FLD = L

The programmer should:

- a. change FLD to BITS;
- b. reorder the arguments; and
- c. change I to I + 1

For example:

FLD(5,6,A)

should be translated to:

BITS(A,6,6)

Or, the programmer could insert the statements:

```
INTEGER FLD
FLD(I,J,A)=BITS(A,I+1,J)
```

at the beginning of the program unit.

- FLD = R

This results in the same action as the L option except for the third step:

- a. change FLD to BITS
- b. reorder the arguments
- c. change I to 36-I

For example:

FLD(5,6,Z)

should be translated to:

BITS(Z,31,6)

or:

```
INTEGER FLD
FLD(I,J,A)=BITS(A,36-I,J)
```

could be inserted at the beginning of the program unit.

- FLD = T Used only for optimization and should be deleted.
- FLD = Q Used only for optimization and should be deleted.
- FLD = ABS Causes all nonconstants in the bit location and length fields to be passed as absolutes. These nonconstants should be made absolute.

For example:

FLD(1,5,C)

should be translated to:

BITS(C,ABS(1) + 1,5)

■ DATA Option:

- DATA=SHORT
- DATA=IBM

These options are used in FORTRAN V to permit partial initialization of arrays. To accomplish the same effect in ASCII FORTRAN, an implied DO-loop is required or else a warning message will be issued.

For example:

```
COMPILER (DATA=SHORT)
DIMENSION A(10)
DATA A/1.,2.,3./
```

should be translated to:

```
DIMENSION A(10)
DATA (A(I),I=1,3)/1.,2.,3./
```

■ 1110=OPT Option:

This invokes a code reordering algorithm. The corresponding ASCII FORTRAN option is U1110=OPT.

■ Other Options:

- ADR=IND
- LIST=DEF
- DIAG=N
- EQUIV=CMN
- MATH=A
- PROP=DPON
- PROP=DPOF
- PROP=RLON
- PROP=RLOF
- PROP=NONE
- XM=1
- CONT=RFOR

These options can be deleted but the programmer should check the ramifications for the processing of each program. Special coding may be required.

10. Execution of FORTRAN V formatted WRITE statements may indicate the presence of the control character by printing a blank character, so that printing will start with position 2 of a print line. ASCII FORTRAN makes no such indication, and printing will start in position 1 of a print line.
11. FORTRAN V allows the programmer to initialize a whole array with one Hollerith literal in a DATA statement, whereas ASCII FORTRAN requires each element of the array to have a Hollerith literal.
12. FORTRAN V allows the programmer to ENCODE/DECODE beyond the area he has given to the ENCODE/DECODE statement. ASCII FORTRAN may give a warning diagnostic and may not ENCODE/DECODE beyond the actual area given or assumed.
13. FORTRAN V allows the unit-number field, the maximum-number-of-records field, and the maximum-record-size field in the DEFINE FILE statement to be an integer constant, an unsubscripted integer variable, or an integer parameter constant. ASCII FORTRAN does not allow it to be an unsubscripted integer variable.
14. FORTRAN V does not rewind an SDF tape file at termination time if a CALL CLOSE has not been done on that file. ASCII FORTRAN does a rewind of the SDF or ANSI tape file if CALL CLOSE or CLOSE has not been done for the file.
15. FORTRAN V checks via an ER FACIT\$ to determine if a unit has been assigned prior to the program execution. If not assigned to the run, it will do an @ASG,T on the unit. ASCII FORTRAN does the ER FACIT\$ but will do an @ASG,A on the unit to determine if a cataloged file already exists for the unit. If it does not exist, an @ASG,T is done on the unit.
16. A FORTRAN V direct access DEFINE FILE statement can be used to extend a direct access file. However, an ASCII FORTRAN direct access DEFINE FILE statement cannot be used to extend a direct access file. In ASCII FORTRAN level 10R1 or higher, the OPEN statement may be used to extend a direct access file.

A.4. Differences in Syntax

1. The FLD function is replaced by the BITS pseudo-function. To change a FORTRAN V reference to the FLD function, to an ASCII FORTRAN reference, the programmer should:
 - a. change FLD to BITS;
 - b. reorder the arguments; and
 - c. change I to I+1

or insert statements such as:

```
INTEGER FLD  
FLD(I,J,A)=BITS(A,I+1,J)
```

To illustrate the first method, the reference:

```
FLD(5,6,A)
```

could be changed to:

```
BITS(A,6,6)
```

2. Four input/output statements in FORTRAN V are expressed differently in ASCII FORTRAN. They are:
 - READ INPUT TAPE *unit,f,list*
 - WRITE OUTPUT TAPE *unit,f,list*
 - READ TAPE *unit,list*
 - WRITE TAPE *unit,list*

To change them to their ASCII FORTRAN equivalents, the programmer should:

- eliminate the words INPUT, OUTPUT and TAPE
- enclose *unit* and *f* (if present) in parentheses and remove the comma before *list*.

For example:

```
READ INPUT TAPE 5,6, A
```

should be changed to:

```
READ (5,6) A
```

and:

```
WRITE TAPE 6, D
```

should be translated to:

```
WRITE (6) D
```


3. A Hollerith constant may be specified in FORTRAN V in the following ways:

- 7HEXAMPLE
- 7REXAMPLE
- 7LEXAMPLE
- 'EXAMPLE'

Of these, only the first and last are permitted in ASCII FORTRAN.

The R and L forms should be changed to the H form. Left and right space fill and shifts may not be significant. Note that ASCII FORTRAN uses four characters per word rather than the six used by FORTRAN V.

4. FORTRAN V considers Hollerith constants to be typeless, whereas ASCII FORTRAN considers them to be of type character. This can cause conflicts in which legal statements in FORTRAN V become illegal in ASCII FORTRAN.

For example, the statement IF (AFLG .NE. 3HEND) GO TO 10 is not allowed in ASCII FORTRAN unless AFLG is declared to be of type character. This could invalidate AFLG in other uses.

5. FORTRAN V permits the concatenation of two arithmetic operators if one of them is unary and it follows **, /, or *. However, this syntax is not allowed in ASCII FORTRAN.

For example, the following are permitted in FORTRAN V but not in ASCII FORTRAN:

```
A=4.0/+B
A=4.0*-B
A=4.0*+B
A=4.0**-I
A=4.0**+I
```

To make them acceptable, the programmer should separate the two consecutive operators by enclosing the unary operator and its operand in parentheses.

For example:

```
A=4.0/-B
```

should be translated to:

```
A=4.0/(-B)
```

6. Complex variables in list-directed input/output statements are not enclosed by parentheses in FORTRAN V.

7. In FORTRAN V:

```
DATA A /1./B/2./
```

is legal. In ASCII FORTRAN, a comma may precede the B:

```
DATA A /1./,B/2./
```

8. In ASCII FORTRAN, the first parameter in a direct access DEFINE FILE statement must be an integer constant.
9. Expressions of the form $A**B**C$ are somewhat ambiguous in interpretation in FORTRAN V. ASCII FORTRAN evaluates the expression as $A**(B**C)$.

Appendix B. ASCII Symbols and Codes

The ASCII symbols and codes are given in Tables B-1 and B-2.

Table B-1. ASCII Control Characters

Octal Code	Symbol	Meaning
00	NUL	Null - may be used as time-fill
01	SOH	Start of heading
02	STX	Start of text
03	ETX	End of text
04	EOT	End of transmission
05	ENQ	Inquiry. <i>Who are you?</i>
06	ACK	Acknowledgment. <i>Yes.</i>
07	BEL	Bell. Human attention is required.
10	BS	Backspace.
11	HT	Horizontal tabulation.
12	LF	Line feed.
13	VT	Vertical tabulation.
14	FF	Form feed.
15	CR	Carriage return.
16	SO	Shift out. Nonstandard code follows.
17	SI	Shift in. Return to standard code.
20	DLE	Data link escape.
21	DC1	Device control for turning on auxiliary device.
22	DC2	Device control for turning on auxiliary device.
23	DC3	Device control for turning on auxiliary device.
24	DC4	Device control for turning on auxiliary device.
25	NAK	Negative acknowledgment. <i>No.</i>
26	SYN	Synchronous idle.
27	ETB	End of transmission block.
30	CAN	Cancel previous data.
31	EM	End of medium.
32	SUB	Substitute character for one in error.
33	ESC	Escape. For code extension.
34	FS	File separator.
35	GS	Group separator.
36	RS	Record separator.
37	US	Unit separator.

Table B-2. Graphic ASCII Characters

Octal Code	Symbol	Octal Code	Symbol
40	Space	120	P
41	!	121	Q
42	"	122	R
43	#	123	S
44	\$	124	T
45	%	125	U
46	&	126	V
47	Apostrophe	127	W
50	(130	X
51)	131	Y
52	*	132	Z
53	+	133	[
54	Comma	134	\
55	Hyphen	135]
56	Period	136	Circumflex
57	/	137	Underscore
60	0	140	Grave Accent
61	1	141	a
62	2	142	b
63	3	143	c
64	4	144	d
65	5	145	e
66	6	146	f
67	7	147	g
70	8	150	h
71	9	151	i
72	:	152	j
73	;	153	k
74	<	154	l
75	=	155	m
76	>	156	n
77	?	157	o
100	@	160	p
101	A	161	q
102	B	162	r
103	C	163	s
104	D	164	t
105	E	165	u
106	F	166	v
107	G	167	w
110	H	170	x
111	I	171	y
112	J	172	z
113	K	173	
114	L	174	
115	M	175	
116	N	176	Tilde
117	O	177	Delete

Appendix C. Programmer Check List

C.1. General

This list is not exhaustive. Its purpose is to point out the more commonly encountered programming errors for the novice programmer and, therefore, help avoid unnecessary bugs.

C.2. Language Errors

The simplest bugs are straightforward language use errors. These are discovered by the compiler during compilation and are flagged in the listing. The programmer could let the compiler do the checking for this type of error and continually change the program until no more errors are flagged. Alternatively, machine time and money could be saved by manually doing a quick check of the program for the following common errors:

1. Do the main program and subprograms contain statements in the following order?
 - COMPILER statement
 - Program declaration statements
 - In a main program, a PROGRAM *pgm* statement is optional.
 - In a subprogram, one of the three program declaration statements must be the first statement of the program unit:
 - a. type FUNCTION $f(a_1, a_2, \dots, a_n)$
 - b. SUBROUTINE $s(a_1, a_2, \dots, a_n)$
 - c. BLOCK DATA [*sub*]
 - IMPLICIT statements
 - Explicit Type statements
 - PARAMETER statements
 - DIMENSION statements

- COMMON statements
- BANK statements
- EQUIVALENCE statements
- EXTERNAL statements
- INTRINSIC statements
- SAVE statements
- DATA statements
- NAMELIST statements
- FORMAT statements
- Statement functions
- Executable statements
- The END statement

Observance of this ordering will prevent the omission or misplacement of necessary program statements.

Some deviation from this order will be accepted. Before changing the order, the user should refer to the section of this manual that specifically discusses the statement in question.

2. Are bodies of source statements contained in positions 7 through 72?

- Do comment lines contain a "C" or "*" in column 1?
- Do continuation lines contain a continuation character in position 6 and blanks in columns 1 through 5?
- Are statement labels contained in positions 1 through 5?

3. Are all arrays dimensioned?

If not, the program may attempt to treat references to the array as function calls. Such errors may show up as undefined references when the program is collected. If an array is not dimensioned and considered as a function by the compiler, its name will appear in the EXTERNAL REFERENCES listing produced by the compiler.

4. Do any array references contain subscripts which are out of bounds?

5. Do all statement labels referenced in GO TO statements exist in the same program unit as the reference?

6. Do function subprograms contain a statement assigning a value to the function?

Since a function reference is used as a value, a value must be assigned to it somewhere in the subprogram.

7. Are DO-loop termination statements placed properly so that only the desired statements are encompassed?
8. Does the number of right parentheses match the number of left parentheses in FORMAT statements, etc?

C.3. Techniques

In addition to language errors, execution errors can be avoided by employing certain coding techniques. Some guidelines for this follow.

1. Are there any repeated sets of code which could be converted to subroutine references?
2. Can the program be broken down into a series of smaller subprograms? If so, does the number warrant the use of COMMON storage?
3. What corrective or diagnostic actions are taken if:
 - incorrect data is read in?
 - numeric overflow, underflow, or divide fault occurs?

A good echo check (to verify that the data read in is what is wanted) is to insert a list-directed output statement immediately following the READ statement (to write the input values with some explanatory text). What programmers think they are reading in may not have any resemblance to what they are actually reading.

For example:

```
PRINT *, 'READ X,Y,I AS: ', X,Y,I
```

4. Insert abundant comments into the code. They help the user to remember what is actually done in a particular coding section and what else is needed by this code. Comments may be particularly helpful at the beginning of a program unit which is used often. Such comments might contain the following information:
 - What is the program designed to do?
 - What input data does it need, if any? In what format and from what device?
 - What subprograms, if any, does the subprogram use? What do these subprograms do?
 - What tapes, files, discs, etc., are used by the program? What are their unit numbers?
 - What forms of data failure can occur during execution of this program? What action is taken in each such case?
5. Use the blocking statements (block IF, ELSE IF, ELSE, and END IF) instead of GO TO statements whenever possible, to make the code more understandable.

Appendix D. Diagnostic Messages

This appendix lists the diagnostic messages (errors, warnings, and nonstandard usages) that may be issued during compilations of ASCII FORTRAN programs. The notation *xx* indicates that situation-dependent data will be filled into the message when the error occurs. The reader should also note that Appendix I contains messages which may be issued during Checkout (C option) runs, and Appendix G has messages at the end which may be issued at run time by the I/O handler.

Number	Message and Brief Description
0001	COMPILER CANNOT ASSIGN SPILL FILE The compiler assigns the temporary file PSF\$ to be used to hold various pieces of information needed during the compilation which will not all fit into main storage. For some reason the compiler cannot assign this file.
0002	I/O ERROR ON COMPILER SPILL FILE <i>xx</i> Some kind of I/O error has occurred on the compiler spill file.
0003	COMPILER SPILL FILE LIMIT EXCEEDED The user program is so large that the compiler has run out of spill file space. The program either has too many variables and equivalences, or, it has too many executable statements.
0004	COMPILER LIMIT OR ERROR <i>xx</i> ENCOUNTERED An unrecoverable internal error has occurred within the compiler. Either some internal limit such as the number of data names is exceeded, or there is an internal error. This event should be reported to the user's local Systems Analyst who may fill out a SUR (Software User Report) to send to Sperry Univac for analysis.
0005	TOO MANY EXTERNAL REFERENCES The external reference table is full. All references to additional external names will be deleted.

0006 DATA SIZE EXCEEDED *xx* LOCATION LIMIT

The total D-bank size allowed for a program is 65,535 decimal words if the O option was *not* used on the compilations. This message will come out with 65,535 if the user's local declarations or any one of the user's common blocks goes over 65,535 words in size without the O option on the compilation. No program can exceed 262,143 words of address space at any one time. This message will come out with 262,143 if the local declarations or any one of the common blocks goes over 262,143 words in size. Programs can go over these sizes if the multi-banking features of ASCII FORTRAN are used. See the BANK and COMPILER statements, and Appendix H.

0007 TOO MANY PROGRAM BLOCKS FOR OPTIMIZATION

The complexity of the program exceeds the ability of the optimization phase to completely analyze it. As much of the program as possible will receive global optimization. The rest of the program will receive only limited local optimization. The programmer should consider simplifying the program by reducing the number of branches (IF, GO TO, DO statements) or by breaking the program into several subprograms.

0008 TOO MANY PROGRAM VARIABLES FOR OPTIMIZATION

There are more variables in the program than can be handled by the optimization phase of ASCII FORTRAN. It is assumed that the most frequently used variables are in the most deeply nested loops, and usage of these will be optimized on a global basis. Expressions involving other variables will only be optimized on a local basis.

0009 TOO MANY USER ENTRY POINTS

The Entry Point Table is full. This entry point and all subsequent user entry points are not inserted. The user should drop some entry points by eliminating some subroutines or ENTRY statements in his source element.

0010 INITIALIZATION OF *xx* IS NOT POSSIBLE

Initialization is not possible for variables or array elements which have addresses over 65K decimal. This occurs on local variables once their accumulated size adds up to this much, or on individual COMMON groups adding up to this much. For example:

```
DIMENSION A(2,33000), B(2,33000)
DATA A(1,1), B(1,1) /1.,2./
```

One of the above initializations would be bad. However, the following would work:

```
DIMENSION A(2,33000), B(2,33000)
COMMON A
DATA A(1,1), B(1,1) /1.,2./
```

Therefore, the solution to this problem is to make smaller COMMON groups.

0011 USED_NUM_INTS GT MAX_NUM_INTS

There are more intervals in the program than can be handled by the information gathering portion of the optimization phase.

0012 USED_NUM_LEVELS GE MAX_NUM_LEVELS

The number of levels resulting from the interval analysis algorithm has exceeded the maximum number of levels that can be handled by the optimization phase.

0015 LASTIBLK GT MAX_NUM_INTS

During the information gathering portion of the optimization phase, the newest initialization block created was assigned an interval number greater than the maximum number of intervals. Global optimization is unable to continue. Local optimization will be attempted.

0016 MISSING END STATEMENT, NEXT PROGRAM UNIT ASSUMED EXTERNAL

1. A BLOCK DATA program is missing an END statement. (BLOCK DATA programs cannot have internal subprograms.)
2. An END was missing from an external program unit and a BLOCK DATA program followed. (BLOCK DATA programs cannot be internal subprograms.)

0017 COMPILER ERROR - TEXT CHAIN ENDS PREMATURELY

A text entry with a zero forward link was encountered before an END text entry was found. This most likely occurred during code generation.

0018 MCORE FAILURE - COMPILER TERMINATED

Optimization (V or Z option) required extra table space. An ER to MCORE\$ was attempted but failed, since the user's system did not have enough available storage. The compiler is terminated. This message may also appear if table space for storage map (D, L, or Y options) or INFO-010 table generation (F or C options) is not available.

0019 PROGRAM UNIT HAS NO EXECUTABLE STATEMENTS

The program unit (main program, external subprogram, or internal subprogram) that just terminated has no executable statements. If the program unit was a subprogram, then the code generated will simply cause it to return control to the caller. If the program unit was a main program, then the code generated will simply consist of calls to FINIT\$ (initialization routine) and FEXIT\$ (termination routine).

0020 MCORE FAILURE - LOCAL OPTIMIZATION ATTEMPTED

If an MCORE failure occurs while attempting to obtain the initial storage for global optimization tables, local optimization will be attempted.

0021 SOURCE LINE CONTAINS MORE THAN 80 COLUMNS

If columns 81-132 appear in the source line, the line is truncated to 80 columns, and is then accepted by the compiler. If more than 132 columns appear in the source line, the line is rejected. In either case, the user should edit the source line so that it is 80 columns or less.

0022 ROR\$ CALL ERROR, BAD ROR PACKET ENCOUNTERED

ROR detected bad contents in a ROR packet. The compilation is terminated.

0023 I/O ERROR DURING ROR\$ CALL, I/O STATUS = *xx*

An I/O error occurred during a ROR call. The I/O status is printed out as a decimal integer. The compilation is terminated.

0024 ROR ERROR, *xx* STATUS = *xx*

1. ROR error, ER-PFWL\$ status = *xx*

A start ROR call received a bad status from an ER PFWL\$ request. The status *xx* is printed out as a decimal integer. The compilation is terminated.

2. ROR error, I/O status = *xx*

An end ROR call received an I/O error. The status *xx* is printed out as a decimal integer. The compilation is terminated.

0025 ROR TABLE WRITE ERROR, *xx* STATUS = *xx*

1. ROR Table Write error, I/O status = *xx*

An ROR Table Write request resulted in an I/O error. The status *xx* is printed out as a decimal integer. The compilation is terminated.

2. ROR Table Write error, ER-PFI\$ status = *xx*

An ROR Table Write request resulted in a bad status from an ER PFI\$ request. The status *xx* is printed out as a decimal integer. The compilation is terminated.

0101 DELIMITER IS MISSING BEFORE *xx*

A required delimiter is missing at the indicated point in the statement. In some cases, the delimiter is inserted (for example, a missing comma after a DATA value list). In other cases, the statement is deleted.

0102 DELIMITER *xx* IS USED INCORRECTLY

The indicated character appeared in a position where it is not permitted. The character may be ignored (for example, an excess comma in a FORMAT statement), or it may cause the statement to be deleted.

0103 *xx* IS INCORRECT OUTSIDE LITERAL CONSTANT

The character printed occurs outside a literal constant, but is not a letter, digit, currency symbol, or FORTRAN delimiter. The character is deleted from the statement; the effect is as if the character were a blank.

0104 NUMERIC CONSTANT *xx* IS OUT OF RANGE

This message indicates that a constant is outside the limits for its type. This can indicate an octal constant which exceeds 12 digits; the last 12 digits are used. For an integer, this message indicates that the value cannot be represented in 35 bits; the last 35 bits of the value are used. For a real constant, this message indicates that the exponent is too large (positive or negative); the value is replaced by zero in the statement.

0105 LONG NAME TRUNCATED TO *xx*

The indicated name is more than six characters long. The first six characters of the name are used.

0106 CONSTANT *xx* IS INCORRECT

The indicated constant is incorrectly formatted. For example, 3.0E-1 would receive this message because the exponent is not an integer constant. In this example, 3.0E-1 would be interpreted as ((3.0E0) - 1). This message also applies to character constants which exceed 511 characters in length; for such constants, the last 510 characters are used.

0107 CONSTANT *xx* IS USED INCORRECTLY

The indicated symbol is used where a particular type of constant is required. The error may cause the statement to be deleted, or it may cause just the affected part of the statement to be ignored.

0108 STATEMENT ENDS PREMATURELY

The end of the statement was encountered while the syntax of the statement was incomplete. The statement may be completed in an arbitrary manner, or the statement may be deleted.

0109 STATEMENT CONTAINS EXCESS RIGHT PARENTHESES

The statement contains at least one right parenthesis for which there is no matching left parenthesis. The statement is deleted.

0110 STATEMENT CONTAINS EXCESS LEFT PARENTHESES

The statement contains at least one left parenthesis for which there is no matching right parenthesis. The statement is deleted.

0111 STATEMENT IS TOO LONG

There are more than 1320 significant characters in the statement, excluding the statement label, if any, and the continuation markers in column 6. This error can only occur for statements having more than 19 continuation lines. The statement is ignored.

0112 UNRECOGNIZABLE STATEMENT

The statement cannot be identified as any FORTRAN statement. The statement is ignored.

0113 STATEMENT CONTAINS UNCLOSED LITERAL CONSTANT

A literal constant had not been closed at the end of the last continuation line of this statement. This can be caused by unbalanced apostrophes, missing delimiters (which can result in apostrophes not being treated as quote characters), a Hollerith field with an incorrect character count, or a missing continuation card. The statement is ignored.

0114 END STATEMENT IS MISSING

The end of the source input was encountered without finding an END statement. This can be caused by failure to place an END statement in the program, or by an error in the terminal statement label of a DELETE statement. The second case will also cause error message 0404. In either case, an END statement is assumed by the compiler.

0115 CHARACTERS *xx* FOLLOW LOGICAL END OF STATEMENT

When the syntax scan detected the end of the statement, the source line had not been completely scanned. The statement is compiled as if it had ended where the syntax scan indicated that the statement was complete.

0116 TARGET STATEMENT OF LOGICAL IF IS OF WRONG TYPE

The statement to be conditionally executed as part of a logical IF statement is not executable, or is a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement. The logical IF and its target statement are ignored.

0117 STATEMENT ILLEGAL IN DEBUG PACKET

A second DEBUG statement appears in the program unit. Since only one DEBUG statement is allowed per program unit, the second is deleted.

0118 NAME *xx* IS USED INCORRECTLY

The name is previously defined or used incorrectly. If this is a warning message, the error is ignored. If it is an error message, the statement is deleted.

0119 STATEMENT ILLEGAL OUTSIDE DEBUG PACKET

TRACE ON, TRACE OFF, AT, or DISPLAY statement appears without a preceding DEBUG statement. This statement is deleted.

0120 SPECIFICATION FOLLOWS DEFINE OR DATA STATEMENT

This specification statement (type, DIMENSION, etc.) follows a statement function definition or a DATA statement. The specification statement is processed as if no error had occurred.

0121 SPECIFICATION FOLLOWS EXECUTABLE STATEMENT

This specification statement follows an executable statement. Specification statements following an ENTRY statement will be flagged with a warning message indicating nonstandard usage. In all other cases, an error will be issued and the erroneous specification statement will be deleted.

0122 PROGRAM TRUNCATED AT END STATEMENT

The only lines that may follow an END statement are comments, START and STOP EDIT statements, FUNCTION, SUBROUTINE, and BLOCK DATA statements, which also signal the start of the source of another program unit. If these rules are not followed, an error 122 is issued and all remaining source is ignored.

0123 AT STATEMENT MISSING IN DEBUG PACKET

An AT statement is required if executable statements follow the DEBUG statement. The executable statements after a DEBUG statement and before any AT statement are processed but can never be executed.

0124 xx FIELD IS REPEATED

0125 STATEMENT ILLEGAL IN BLOCK DATA SUBPROGRAM

An executable statement or other statement not permitted in a BLOCK DATA subprogram has been encountered. The statement is deleted.

0126 STATEMENT ILLEGAL IN FUNCTION SUBPROGRAM

0127 CONTINUED STATEMENT HAS NO INITIAL LINE

The indicated statement begins with a continuation line. The initial line is assumed to be blank. If a label appears in columns 1 through 5 of the continuation line which begins the statement, it is taken to be the statement label. This is the only situation in which the label field of a continuation line will be interpreted by the compiler.

0128 COMPILER STATEMENT OPTION xx CONFLICTS WITH PREVIOUS OPTIONS

Certain options of the COMPILER statement are incompatible; for example, ARGCHK=ON is not compatible with ARGCHK=OFF. This error may also be received if the DATA=AUTO, DATA=REUSE, or PROGRAM=BIG options are misplaced in the source. Each of the three options should appear at the beginning of the source for the compilation.

0129 EMPTY STRING INTERPRETED AS STRING OF ONE BLANK

Null strings are not permitted. They are changed internally to a string of one blank.

0130 SYNTAX ERROR IN CALL STATEMENT

There is a syntax error in the argument list to the subprogram being called in a CALL statement, or, there are nonblank characters following the argument list.

0131 SYNTAX ERROR IN CALL STATEMENT AT *xx*

Something other than a left parenthesis followed the subprogram name in a CALL statement.

0201 IMPLICIT STATEMENT IS OUT OF CORRECT ORDER

The IMPLICIT statement must precede any specification or executable statements. The IMPLICIT statement is accepted, but will affect only variables which have not yet appeared in any statement.

0202 LENGTH SPECIFICATION *xx* IS INCORRECT

The length specification is not a positive integer constant or is not permitted for the requested type. The optional length for the requested type is used. The length used for type CHARACTER is 1.

0203 TYPE SPECIFICATION *xx* IS INCORRECT

The type keyword in the IMPLICIT statement is not a recognized FORTRAN data type. The incorrect type specification is ignored.

0204 LETTER SPECIFICATION *xx* IS INCORRECT

The indicated character is not a valid IMPLICIT range limit. The character is ignored. Thus REAL(I-*) is the same as REAL(I).

0205 LETTER SPECIFICATION *xx* OCCURS PREVIOUSLY IN IMPLICIT

An IMPLICIT type was previously specified for the indicated letter. The new type specification for the indicated letter is ignored.

0206 STMT. ORDER PROBLEM FOR *xx*

A DATA statement intervenes between the type specification for the variable and its preceding dimension specification. The DATA statement does not necessarily refer to the constant. The type specification for this variable is ignored. See Appendix A for a statement ordering description.

0301 PARAMETER NAME *xx* IS PREVIOUSLY DEFINED

The statement redefines the indicated parameter constant. The attempted redefinition is ignored.

0302 PARAMETER NAME *xx* IS INCORRECT

The symbol following the PARAMETER keyword is not a simple variable name. The PARAMETER definition is ignored. This message is also produced when a formal parameter name in a FUNCTION, ENTRY, or SUBROUTINE statement is incorrect (for example, a constant). The incorrect formal parameter is deleted.

0303 PARAMETER EXPRESSION IS NOT CONSTANT VALUED

A PARAMETER expression can only contain constants, previously defined parameter constants, and references to certain intrinsic functions. The PARAMETER definition does not satisfy this constraint and is ignored.

0304 PARAMETER EXPRESSION MAY GIVE QUESTIONABLE RESULTS

Since parameter constants have a type associated with them (FORTRAN 77), assigning a parameter constant a typeless constant can produce unpredictable results when the parameter constant is used in an expression. This comes about because the internal representation of the typeless constant (a 1-word string) may not conform to the internal representation expected by the operator.

0401 INCLUDE PROCEDURE CANNOT BE FOUND

The procedure name specified in the INCLUDE statement could not be found. The INCLUDE statement is ignored. This message will also occur if an I/O error is encountered while searching for an INCLUDE procedure.

0402 NESTED INCLUDE IS ILLEGAL

An INCLUDE statement has been encountered within the text copied by an INCLUDE statement. This is not allowed. The nested INCLUDE is ignored.

0403 INPUT ERROR *xx* ON INCLUDE FILE

The indicated I/O status was received on an ER IOW\$ while reading or opening an INCLUDE procedure. If the error occurs while opening the procedure, the INCLUDE is ignored and error 0401 is also generated. If the error occurs while reading the body of the procedure, the INCLUDE procedure is immediately terminated, as if the end-of-file sentinel had been encountered. If, instead of an I/O status, the error read "NOT-PF", then the file is not a program file.

0404 TERMINAL LABEL FOR DELETE CANNOT BE FOUND

The end of the source input was reached without finding the statement label which terminates the active DELETE statement. The DELETE is automatically terminated at the end of the source input.

0405 CORRECTION IMAGE SEQUENCE ERROR

A source correction line is out of sequence or is incorrect. The correction line was just printed in a previous message, for example, 'OUT OF SEQ -15'.

0406 PROCEDURE NAME *xx* IS INCORRECT

The indicated INCLUDE procedure name is invalid. The INCLUDE statement is ignored.

0407 INCLUDE OPTION *xx* IS INCORRECT

The indicated characters were encountered where the LIST option was expected. The LIST option is assumed.

0408 EDIT OPTION *xx* IS INCORRECT

The indicated option for a START EDIT or STOP EDIT statement should be either SOURCE, CODE, or PAGE. If the statement is START EDIT, the SOURCE option is assumed. If the statement is STOP EDIT, CODE is assumed.

0409 ERROR TYPE *xx* ON INPUT

The indicated I/O status was received from an ER IOW\$ on input from the source input stream. The compilation is terminated without producing an updated source element.

0410 INPUT ERROR IN OPEN

An I/O error has occurred in opening the source input stream. The compilation is discontinued immediately.

0411 PFI\$ ERROR IN CLOSE

0412 INCLUDE FILE CANNOT BE ASSIGNED

The explicitly named INCLUDE file could not be assigned. The INCLUDE statement is deleted.

0501 VARIABLE *xx* HAS BEEN PREVIOUSLY DEFINED

The indicated variable has already had a type specified in a type statement. The new specification for this variable is ignored. This message also occurs in other cases where a name is doubly defined.

0502 LENGTH SPEC. *xx* IS INCONSISTENT OR INCORRECT

The specified type does not permit a length specification or does not permit the length that is specified. The length specification is ignored.

0503 CHARACTER **(*)* NOT ALLOWED FOR *xx*

Dynamic character string arguments are not allowed for programs with old mode on (COMPILER(STD=66)), or on names which are not function names, argument names, or parameter constants.

0504 CHARACTER **(*)* NOT ALLOWED WITH STD=66

See the explanation for 0503.

0601 DIMENSION OR SUBSTRING *xx* IS NOT TYPE INTEGER

The constant specified for the array's dimensions or the character item's substring start or end position is not type integer.

0602 ARRAY *xx* HAS MORE THAN SEVEN DIMENSIONS

0604 DIMENSION *xx* IS NOT A SCALAR, ARGUMENT, OR IN COMMON

0605 ARRAY *xx* IS PREVIOUSLY DEFINED

0606 VARIABLE DIMENSION IS NOT PERMITTED FOR *xx*

The indicated variable has been specified with variable dimensions, but is not an argument of the subroutine.

0607 ASTERISK NOT PERMITTED AS DIMENSION BOUND FOR *xx*

An asterisk has been specified as a dimension in which the array is not a dummy array or the asterisk is not the upper dimension of the last dimension.

0608 ARRAY *xx* IS TOO LARGE

The number of array elements or the total word size of the array exceeds 262,143. The array bounds must be reduced.

0609 DIMENSION BOUND EXPRESSION IS NOT TYPE INTEGER FOR *xx*

After calling the expression routine to evaluate a dimension the result type was not integer.

0610 LOWER DIMENSION BOUND GREATER THAN UPPER DIMENSION BOUND FOR *xx*

0612 SUBSCRIPT OR SUBSTRING NOT INTEGER CONSTANT EXPRESSION FOR *xx*

The array name specified in the DIMENSION or EQUIVALENCE statement contains subscript references that must be integer constant expressions or the character item contains substring references that must be integer constant expressions.

0613 DIMENSION VALUE INDETERMINABLE AT COMPILE TIME FOR *xx*

A program contains an array with a dimension expression that cannot be calculated at compile time because the ASCII FORTRAN compiler has been configured without the mathematical common banks (for example, 3**3 cannot be computed) or else the expression is out of range (for example, 200,000,000 ** 200,000,000).

0614 DIMENSION BOUND *xx* IS NOT DUMMY ARGUMENT OR IN COMMON

The indicated name appears in a dimension bound or in a dimension bound expression for an adjustable array declaration. The name cannot be a local variable. It must be a dummy argument, a variable appearing in a common block, or a parameter constant name.

0701 VARIABLE *xx* APPEARS IN PREVIOUS COMMON STATEMENT

0702 ARGUMENT *xx* APPEARS IN COMMON OR EQUIVALENCE STATEMENT

The indicated name is not permitted in a COMMON block or in equivalence. The name is ignored.

0703 *xx* MAKES COMMON TOO LARGE

Inclusion of the indicated variable causes a COMMON block to exceed 262,143 words. The assignment of addresses to this variable and those following it in the COMMON block will be incorrect.

0704 MAXIMUM NUMBER OF COMMON BLOCKS EXCEEDED

A maximum of 247 common blocks were specified in all program units of a FORTRAN compilation.

0801 EQUIVALENCE MEMBER *xx* LINKS COMMON BLOCKS

The indicated item is in a COMMON block and is equivalenced to an item in another COMMON block. The equivalence for the item is ignored.

0802 EQUIVALENCE MEMBER *xx* IS THE ONLY MEMBER OF A SET0803 EQUIVALENCE MEMBER *xx* IS INCONSISTENT

This item is equivalenced in a way incompatible with prior equivalences. The inconsistent equivalence is ignored.

0804 EQUIVALENCE MEMBER *xx* REORDERS A COMMON BLOCK

The indicated item is equivalenced in a way which conflicts with prior EQUIVALENCE and COMMON statements. The incorrect equivalence relationship is ignored.

0805 EQUIVALENCE MEMBER *xx* EXTENDS COMMON INCORRECTLY

The indicated item causes a COMMON block to be extended backward. The equivalence is ignored.

0806 *xx* SUBSCRIPT OUT OF RANGE IN EQUIVALENCE

A subscript for the indicated array is out of range. The variable is treated as a scalar.

0807 *xx* WRONG NUMBER OF SUBSCRIPTS IN EQUIVALENCE

The indicated array appears with more or fewer subscripts than it should have. Missing subscripts are assumed to be one; excess subscripts are ignored.

0808 EQUIVALENCE MEMBER *xx* MUST BEGIN ON WORD BOUNDARY

The item indicated is equivalenced to a character item that begins on a nonword boundary. The item is assigned storage on the previous word border.

0809 *xx* MAKES EQUIVALENCE GROUP TOO LARGE

The item makes an equivalence group exceed 262,143 words. Storage assignment for variables in the group will be incorrect.

0810 UNDIMENSIONED NAME *xx* USED IN EQUIVALENCE

The indicated name appears with subscripts in an EQUIVALENCE statement, but is not dimensioned. The subscripts are ignored.

0901 EXTERNAL NAME *xx* IS NOT A SUBPROGRAM NAME

1001 BANK MEMBER *xx* IS PREVIOUSLY DEFINED

1002 BANK MEMBER *xx* IS NOT COMMON OR EXTERNAL NAME

The indicated name appears in a BANK statement, but is neither a COMMON block name nor an external subprogram name. The BANK specification is ignored for this item.

1003 GENERIC FUNCTION *xx* DEINTRINSIFIED VIA BANK STATEMENT

The user has named a generic function in a BANK statement. It is treated as if he had also named it in an EXTERNAL statement. All automatic argument and result typing is lost because it is not considered an intrinsic function.

1004 THE BANK STATEMENT ON *xx* IS MISPLACED

The user has named a function/subroutine in a BANK statement in an internal subroutine, but the function/subroutine was first used in the outer external procedure. If he wants to "bank" the function, he must put the statement in the outer external procedure.

1005 INTERNAL SUBPROGRAMS CANNOT BE BANKED

The user has tried to name an internal subroutine in a BANK statement. This is illegal.

1101 NAMELIST NAME *xx* IS PREVIOUSLY DEFINED

The NAMELIST name has already occurred as a variable, array, or NAMELIST name. The NAMELIST statement is deleted.

1102 NAMELIST LIST REF. *xx* IS INCORRECT

An item in the NAMELIST list is not a variable, array, or array element with constant subscripts, or is a formal parameter of the subprogram. The NAMELIST statement is deleted.

1201 *xx* MAY NOT BE INITIALIZED

The indicated identifier is an argument of the subprogram. Since initial value assignment is not permitted for formal arguments, the DATA statement or initial value specification is ignored.

1202 CONSTANT DOES NOT MATCH TYPE OF VARIABLE *xx*

The initial value is of a type not permitted for the indicated variable. If the message is an error, the assignment is not done. If the message is a warning, the constant is converted to the type of the variable and the assignment is done.

1203 NUMBER OF CONSTANTS EXCEEDS NUMBER OF VARIABLES

The initial value list contains more elements than the list of variables to be initialized. The excess constants are ignored.

1204 NUMBER OF VARIABLES EXCEEDS NUMBER OF CONSTANTS

The list of variables is longer than the initial value list. The excess variables do not receive any initialization.

1205 IMPLIED DO IS TOO COMPLEX - SIMPLIFY

The implied-DO in a DATA statement is too complex, causing the interpreter stack to overflow. The rest of the current variable list is deleted.

1206 REPEAT COUNT ERROR AT *xx*

There is some kind of syntax error on a repeat count in the DATA statement.

1207 OPERAND *xx* INCORRECT FOR DATA STATEMENT

This message means one of two things has occurred:

1. The indicated operand appears in an implied-DO in a DATA statement and is not permitted there. Only constants, parameter constants, and DO-loop index variables are permitted in expressions in an implied-DO in a DATA statement.
2. The operand is an argument to the subprogram and cannot be initialized in a DATA statement.

1208 CHARACTER STRING(S) TRUNCATED IN DATA CONSTANT LIST

The length of one or more character strings in a DATA statement constant list exceeds the storage limits for the variable in the corresponding data list. The character constant is truncated on the right.

1209 SUBSTRING EXPRESSION MUST BE AN INTEGER CONSTANT EXPRESSION

Substring expressions must be integer constant expressions.

1210 OBJECT OF SUBSTRING MUST BE A VARIABLE OR ARRAY ELEMENT

The object of a substring reference when used in a DATA statement must be a scalar character variable or character array element.

1211 SUBSTRING REFERENCE *xx* NOT PERMITTED HERE

1212 MORE THAN ONE INITIALIZATION FOR *xx*

A variable, array element or substring must not be initially defined more than once in an executable program. If two entities are associated, only one may be initially defined in a DATA statement in the same executable program. The initial value received by the entity is unpredictable.

1301 BLOCK DATA STATEMENT IS OUT OF CORRECT ORDER

1302 *xx* IS LOCAL TO BLOCK DATA - IGNORED

The indicated variable appears in a type, DIMENSION, or DATA statement in the BLOCK DATA subprogram, but is not in a common block. No storage is allocated for the variable, and any initial value assigned to it is ignored.

1401 ARGUMENT *xx* IS REPEATED IN ARGUMENT LIST

A format parameter name appears more than once in the parameter list. Only the first appearance is processed.

1402 INTERNAL SUBPROGRAM *xx* SHOULD NOT BE USED AS ARGUMENT

This diagnostic is issued whenever an internal subprogram is passed as an argument to an external subprogram and the DATA=AUTO or DATA=REUSE COMPILER statement options are present. Because of automatic storage stack conventions, an internal subprogram can only be called from the external program unit it is contained in or from another internal contained in the same external as itself.

1403 STATEMENT IS USED IN MAIN PROGRAM

An ENTRY statement is not permitted in a main program. The ENTRY statement is deleted.

1404 ENTRY STATEMENT IS WITHIN RANGE OF DO-LOOP OR BLOCK IF STRUCTURE

An ENTRY statement is not allowed in the range of a DO-loop or in the range of a block IF structure (that is, from an IF (*e*) THEN statement to the corresponding END IF statement). The ENTRY statement is deleted.

1405 CHARACTER AND NON-CHARACTER ENTRY NAMES NOT ALLOWED

A function subprogram contains a combination of character and noncharacter entry or function names. If the function name is type character all entry names must be type character. If an entry name is type character, the function name must be type character.

1406 SUBPROGRAM OR SUBPROGRAM REF. HAS TOO MANY ARGUMENTS

There is a compiler limit of 150 subprogram arguments allowed, in either a subprogram reference (that is, calling a subroutine, or referencing a function in an expression) or a subprogram itself (total number or arguments in all SUBROUTINE, FUNCTION, or ENTRY statements in the subprogram). There is also a limit of 63 character subprogram arguments allowed.

1407 ENTRY, FUNCTION, OR SUBROUTINE REQUIRES A NAME

The statement requires a name and none was found. The statement is analyzed for further errors, then deleted. The routine is typed as a subroutine or function if the error is in a SUBROUTINE or FUNCTION statement.

1408 FUNCTION NAME MAY NOT BE ASSIGNED A VALUE

1410 A PREVIOUS RECURSIVE CALL IS IN ERROR

1411 FUNCTIONS AND ARGUMENTS ARE INCONSISTENT WITH COMPILER OPTION DATA=REUSE

The COMPILER statement option DATA=REUSE means that the routine when entered will reuse the automatic storage space of the calling program. This means that the program can never be returned to, since all register save areas are destroyed, so a function makes no sense. Also, arguments are passed in the automatic storage stack, and so passing arguments is not possible.

1412 SUBSTRING EXPRESSION MUST BE TYPE INTEGER

The expressions e_1 and e_2 in a substring reference ($e_1:e_2$) must both be type integer.

1413 INCORRECT SUBSTRING REFERENCE

A substring reference must have the format ($e_1:e_2$) immediately following a character scalar variable name or a character array element name, where e_1 and e_2 are integer expressions.

1414 CONSTANT USED AS SUBSTRING EXPRESSION OUT OF RANGE

The values e_1 and e_2 in a substring reference ($e_1:e_2$) must satisfy the following relational: $1 \leq e_1 \leq e_2 \leq len$, where e_1 and e_2 are integer expressions, and len is the number of characters in the character scalar variable or character array element immediately preceding the format ($e_1:e_2$). If either e_1 or e_2 is a constant expression, then checks are made at compilation time to see that the preceding relational expression is true.

1415 SUBSTRING VALUE OUT OF RANGE FOR EQUIVALENCE ITEM *xx*

The substring *start* and substring *end* values must satisfy the following relational expression: $1 \leq start \leq end \leq len$, where len is the declared character length of the item.

1416 CHARACTER SUBSTRING REFERENCE INVALID FOR EQUIVALENCE ITEM *xx*

The substring reference for the item in equivalence must be on a character scalar or character array element.

1502 STMT. FUNCTION DEF. HAS TOO MANY ARGUMENTS

The statement function definition has more than the compiler limit of 150 arguments. The statement is deleted.

1503 STMT. FUNCTION DEF. ARGUMENT *xx* IS INCORRECT

The indicated item appears where a formal argument is expected, and either is not a name or is the statement function name. The statement is deleted.

1504 STMT. FUNCTION NAME *xx* IS PREVIOUSLY DEFINED

The statement function name has been previously used in a way which precludes its use as a statement function name. The statement is deleted.

1505 STMT. FUNCTION PARAM. *xx* IS NOT USED IN DEF.

1601 STATEMENT LABEL IS INCORRECT

Columns 1-5 of the following source line contain at least one character other than a space or a digit, and the line is not a comment line. The contents of columns 1-5 are ignored.

1602 STATEMENT LABEL *xx* IS PREVIOUSLY DEFINED

Statement label already identifies a previous statement. This label is deleted.

1603 LABELLED BLANK LINE TREATED AS CONTINUE STATEMENT

1604 COLUMNS 1-5 OF CONTINUATION LINE SHOULD BE BLANK

Nonblank characters appear in columns 1-5 of a continuation line. They are ignored.

1605 STATEMENT SHOULD HAVE A LABEL

FORMAT statements and statements following a GO TO statement should be labeled. The statement is processed, but cannot be referenced or executed. This error may be caused by error 1602.

1606 LABEL *xx* REFERENCED OUTSIDE DEBUG PACKET

1608 FORMAT LABEL *xx* IS USED INCORRECTLY AS A BRANCH POINT

The label on the FORMAT statement was previously used incorrectly as the target of an IF or GO TO. Bad code may be generated for the statement.

1609 LABEL ON THIS NON-EXECUTABLE STMT. WAS PREVIOUSLY REFERENCED

Previous references to this label are in error. Note that any statement may be labeled, but only labeled executable statements (except for DEFINE FILE, ELSE, and ELSE IF) and FORMAT statements may be referenced by the use of statement labels. There is one exception: the DELETE statement may refer to any statement label.

1610 STMT. REFERENCES NON-EXECUTABLE STMT. LABEL *xx*

This statement is in error, since it refers to a label which may not be referenced. Note that any statement may be labeled, but only labeled executable statements (except for DEFINE FILE, ELSE, and ELSE IF) and FORMAT statements may be referenced by the use of statement labels. There is one exception: the DELETE statement may refer to any statement label.

1701 WRONG NUMBER OF SUBSCRIPTS FOR ARRAY REFERENCE *xx*

A reference to the indicated array has the wrong number of subscripts. The statement is deleted.

1702 SUBSCRIPTS REQUIRED FOR ARRAY REFERENCE *xx*

The indicated array name appears in a context which requires a scalar reference. The statement is deleted.

1703 SUBSCRIPT EXPRESSION IS NOT CONSTANT

Subscripted references in DISPLAY and NAMELIST statements must have constant subscripts. The statement is deleted.

1704 ARRAY *xx* IS NOT DIMENSIONED1705 SUBSCRIPT IS OUT OF RANGE OF ARRAY *xx*

The constant subscript is out of range. The subscript is accepted as written except for subscript errors occurring in data initialization. In this case the data initialization is not done.

1706 SUBSCRIPTED REFERENCE *xx* NOT PERMITTED HERE1707 SUBSCRIPT *xx* IS OF THE WRONG TYPE

A logical, complex, or character expression has been used as a subscript. The statement is deleted.

1708 REF. TO VARIABLE DIMENSIONED ARRAY *xx* BEFORE ENTRY STMT

A reference has been made to an array which has been declared with variable dimensions but has not yet been described in a SUBROUTINE or ENTRY statement. This may result from omitting declarations for parameter constants, then using these omitted constants as dimension specifications in a DIMENSION statement. Arrays so dimensioned appear to be variable-dimensioned.

1709 DIMENSION BOUNDS FOR *xx* CAUSE COMPILER LIMIT OVERFLOW

When calculating the virtual origin value ($\text{location_counter_offset} - \text{sum_of_multipliers} + \text{byte_offset}$) for an array, the value exceeded the limit allowed by the compiler. The dimensions for the array must be changed.

1710 RELATIVE ADDRESS FOR *xx* EXCEEDS LIMIT

This message is usually the result of an array reference such as ARR(70000) where the constant subscript value exceeds 65,535 words, the array is dimensioned less than 65,536 words, and the O option is not present on the @FTN control card. This message actually will appear whenever an instruction is built by code generation and the address portion is truncated when filled into the 16- or 18-bit u-field of an instruction requiring relocation.

1801 EXPRESSION IS TOO COMPLEX - SIMPLIFY

The current expression is too complex for the expression scanner. It should be broken into two or more statements. This error can be caused by nesting array, function, and statement function references too deeply.

1802 COMPLEX CONSTANT PARTS ARE NOT THE SAME LENGTH

One of the components of a COMPLEX constant is REAL and the other is double precision. The shorter component is converted to double precision and the resulting constant is COMPLEX*16.

1803 COMPLEX CONSTANT PARTS ARE NOT BOTH REAL

1804 CONSTANT EXP. EVALUATION PRODUCES *xx* ERROR

The combination of two constants has resulted in an arithmetic fault. The type of fault is indicated in the message. If the fault is "UNDRFLOW", the result is set to zero. Otherwise, the constant valued expression is not evaluated.

1806 SUBPROGRAM REF. *xx* HAS WRONG NUMBER OF ARGUMENTS

The indicated function or subroutine was first referenced with a different number of arguments. This statement is processed normally, unless the function is a MAX or MIN function, in which case the call is ignored. If the function was a FORTRAN-supplied function and referenced with no arguments, it will be treated subsequently as a scalar. (See 7.2.1.)

1807 STMT. FUNCTION REF. *xx* HAS WRONG NUMBER OF ARGUMENTS

The number of arguments in the statement function reference does not match the number in the definition. The statement is deleted.

1808 SUBROUTINE NAME *xx* IS USED INCORRECTLY

A subroutine is being referenced as a function. The statement is deleted.

1809 ARGUMENT *xx* IS OF THE WRONG TYPE1810 LOGICAL OPERAND *xx* IS OF THE WRONG TYPE

The expression contains a logical operator (.AND., .OR., .NOT.) with a nonlogical operand. The statement is deleted.

1811 ARITHMETIC OPERAND *xx* IS OF THE WRONG TYPE

The expression contains an arithmetic operator (+, -, *, /, **) with a nonarithmetic operand. The statement is deleted.

1812 RELATIONAL OPERAND *xx* IS OF THE WRONG TYPE

The expression contains a relational operator (.LT., .LE., .NE., .EQ., .GE., .GT.) with a logical or complex operand. The statement is deleted.

1813 TYPELESS OPERAND *xx* IS OF THE WRONG TYPE

A typeless operand has been used in an expression with a complex or double precision operand. The statement is deleted.

1814 CHARACTER OPERAND *xx* IS OF THE WRONG TYPE

The expression contains a concatenation operator with a noncharacter operand. The statement is deleted.

1815 UNARY OPERATOR *xx* IS USED INCORRECTLY

The indicated unary operator appears in a context which requires a binary operator, or it follows another unary operator. The statement is deleted.

1816 BINARY OPERATOR *xx* IS USED AS A UNARY OPERATOR

The indicated operator appears where an operand is required. The statement is deleted.

1817 *xx* IS NOT A SUBROUTINE NAME

The name following the keyword "CALL" has been previously used in a way which precludes it being a subroutine name. The CALL statement is deleted.

1818 TYPE CONVERSION IS INCORRECT

The expression in a RETURN statement is not type integer. It is converted to an integer, if possible. This message also occurs if the target and expression in an assignment statement are of incompatible types (for example, character and integer). The assignment statement is deleted.

1819 FUNCTION REF. *xx* IS INCONSISTENT WITH COMPILER DEF.

1. A FORTRAN-supplied function name is referred to with the wrong number of arguments or with arguments of the wrong type. The function is assumed to be a user-supplied external function if this is its first occurrence. Error 1826 will also be printed to inform the user of this change.
2. This message may also be emitted if the user uses a generic function name without a local specification statement on the name.

1820 CHARACTER EXPRESSION EXCEEDS ALLOWABLE LENGTH

A character string expression exceeds the compiler limit of 511 characters. The statement is deleted.

1821 *xx* ARGUMENT IS OUTSIDE PERMITTED RANGE

An argument to the indicated FORTRAN-supplied function was incorrect.

1822 *xx* ARGUMENT IS NOT NORMALIZED

An argument to the indicated FORTRAN-supplied function was incorrect.

1823 REF. TO GENERIC NAME *xx* MAY BE INCONSISTENT

This message sometimes will be emitted when the user uses a generic function name as a local variable with no specification statement on the name. The name is treated as a local scalar with default typing.

1825 DIVISION BY ZERO DETECTED IN THIS STATEMENT

Division by constant zero or a parameter constant whose value is zero has been detected in an expression

1826 FUNCTION *xx* IS NOW A USER-SUPPLIED FUNCTION

The first occurrence of a compiler-defined function had the wrong number or type of arguments. The function is assumed to be user-supplied and loses any intrinsic properties.

1827 UNARY OPERATOR '(' FOLLOWS CONSTANT IN EXPRESSION

A left parenthesis cannot follow a constant in any expression syntax. This message could come out in a statement-function expansion. For example:

```
character*(*) sf
sf(arg) = arg(1:2)
print *, sf('1234')
```

This results in the expanded expression: '1234'(1:2) which is not allowed (since only scalars and array elements may precede the (*e1*:*e2*) substring syntax).

1828 INTRINSIC NAME *xx* CANNOT BE USED AS ARGUMENT

Certain intrinsic functions cannot be passed as an argument to another routine because:

1. They are generated as inline functions and have no external entry point to pass and thus cannot possibly be called.
2. They have a calling sequence that is unique to their name, such as LOWERC and UPPERC, and thus cannot be called by a dummy subprogram name.

1901 ASSIGNMENT RESULT *xx* IS INCORRECT

The item on the left-hand side of the assignment is not permitted as the target of an assignment. The statement is deleted.

1902 ASSIGNMENT RESULT *xx* IS AN ACTIVE DO VARIABLE

The indicated item is the index variable of an active DO-loop. The statement is deleted.

1903 ASSIGNMENT RESULT *xx* DOES NOT MATCH EXPRESSION

1904 ASSIGNMENT RESULT *xx* EXCEEDS COMPILER TABLE LIMIT

More than the compiler limit of 128 items appear in the left-hand side of a multiple assignment statement. The statement is deleted.

2001 LABEL REFERENCE *xx* IS INCORRECT

The statement label in an AT statement is missing, or it does not refer to a previous executable statement. The AT statement is deleted. This message can also occur if an *, \$, or & in the actual argument list of a subroutine call is not followed by a statement label, or if a statement label is passed as an actual parameter in a function reference. In this case, the statement is deleted. This message is also generated if the statement label following END= or ERR= clause of an input/output statement has appeared previously on a nonexecutable statement, in which case the statement is deleted.

2002 LABEL REFERENCE EXPRESSION IS INCORRECT

This message can occur for two reasons:

1. A statement of the form RETURN *i* has been encountered in a subprogram which has no asterisks in its parameter list.
2. The *i* specified in a RETURN *i* statement is either negative or too large for the asterisk list.

2003 ASSIGN VARIABLE *xx* IS INCORRECT

2004 LABEL REFERENCE *xx* EXCEEDS COMPILER TABLE LIMIT

2005 STATEMENT LABEL *xx* IS NOT DEFINED

The indicated statement label is referenced, but never appears as the label of a statement. The reference may be deleted or it may be generated as a jump to an undefined address.

2006 KEYWORD *xx* USED AS LABEL REFERENCE IN ARITHMETIC IF STMT.

The IF statement is recognized by the compiler as an arithmetic IF consisting of only the negative branch which is a FORTRAN keyword. The warning is issued because the programmer probably wanted a logical IF. For example:

```
IMPLICIT INTEGER (A-Z)
IF (LOGFCT) RETURN
```

2104 DEBUG FACILITY AND OPTIMIZATION ARE INCOMPATIBLE

The combination of a DEBUG statement and an optimization option (V or Z) is not allowed, since bad code could be generated. Optimization is not equipped to handle debug code.

2105 DEBUG SUBCHK NOT PERMITTED FOR ASSUMED-SIZE ARRAY *xx*

An assumed-size array (such as DIMENSION A(*)) cannot be specified for subscript checking since the size of the array cannot be calculated.

2201 DO VARIABLE IS USED IN AN OUTER DO

The index variable of this DO statement is also the index variable of an earlier unclosed DO-loop. The statement is accepted. The result of executing the DO-loops is unpredictable.

2202 RANGE OF DO CONTAINS NO EXECUTABLE STATEMENTS

There are no executable statements in the indicated DO-loop. The DO statement is ignored.

2203 DO LOOPS ARE INCORRECTLY NESTED

The terminal statement of a DO statement has been encountered, and the terminating statement for a later DO statement has not yet been encountered. All unclosed DO-loops encountered since the closing DO are also closed at this statement.

2204 DO TERMINAL STATEMENT NUMBER IS INCORRECT

The statement label which follows the keyword DO is incorrect. The DO statement is ignored.

2205 DO TERMINAL STMT. NUMBER IS PREVIOUSLY DEFINED

The DO terminal statement label must appear physically after the DO statement in the same compilation unit. The DO statement is ignored.

2207 DO INCREMENT VALUE IS INCORRECT OR INCONSISTENT

The increment value is not an integer, real, or double precision expression or has constant value zero. This message can also occur if the initial, increment, and terminal values are all constants and the sign of the increment value is not consistent with the sign of the difference of the terminal and initial values. The DO statement is ignored.

2208 DO TERMINAL VALUE IS INCORRECT

The DO terminal value is not an integer, real, or double precision expression. The DO statement is ignored.

2209 DO VARIABLE IS SUBSCRIPTED

The DO index variable may not be an array element. It must be a simple integer or real variable. The DO statement is ignored.

2210 DO VARIABLE IS OF THE WRONG TYPE

The DO index is not an integer or real variable. The DO statement is ignored.

2211 DO VARIABLE IS INCORRECT

There is an error in the DO index variable. The DO statement is ignored.

2212 DO INITIAL VALUE IS OF THE WRONG TYPE

The initial value is not integer, real, or double precision. The DO statement is ignored.

2213 DO INCREMENT VALUE IS OF THE WRONG TYPE

The increment value is not integer, real, or double precision. The DO statement is ignored.

2214 DO TERMINAL VALUE IS OF THE WRONG TYPE

The terminal value is not integer, real, or double precision. The DO statement is ignored.

2215 DO TERMINAL STMT. TYPE PREVENTS LOOP COMPLETION

The terminal statement of the DO-loop does not permit execution of the following statement. As a result, the increment and test parts of the DO-loop will never be executed.

2216 I/O LIST CONTAINS REDUNDANT PARENTHESES

An element or sublist in an implied-DO is enclosed in parentheses. The redundant parentheses are ignored.

2217 IMPLIED DO LIST CONTAINS NO MEMBERS

The implied-DO list is empty. The implied-DO is ignored.

2218 IMPLIED DO LIST CONTAINS EXPRESSION OR CONSTANT

The elements of an implied-DO list should be variables, arrays, and array elements. For an input list, the implied-DO is ignored. For an output list, the error is ignored.

2219 IMPLIED DO ELEMENT MUST BE SUBSCRIPTED ARRAY

An element of an implied-DO list used in data initialization is not an array element. The implied-DO is ignored.

2220 SYNTAX ERROR IN IMPLIED DO

2221 DO NESTING EXCEEDS COMPILER LIMIT

The level of DO and implied-DO nesting exceeds the compiler limit of 25. The DO or implied-DO is ignored.

2222 DATA LIST CONTAINS EXPRESSION OR CONSTANT

An element of a DATA statement variable list is not a variable, array element, or array. The list element is ignored.

2223 BLOCK IF NESTING EXCEEDS COMPILER LIMIT

The level of block IF statements exceeds the compiler limit of 25. The block IF statement is ignored. Note that an END IF statement is required to terminate a block IF nesting level (which is started with a block IF statement, that is, IF (*e*) THEN).

2224 END IF, ELSE, OR ELSE IF STMT. IS NOT IN BLOCK IF STRUCTURE

The END IF, ELSE, or ELSE IF statement must have a matching block IF statement (that is, an IF (*e*) THEN statement at the same IF-level) preceding it in the program unit. The END IF, ELSE, or ELSE IF statement is ignored. Note that each block IF statement must have exactly one corresponding END IF statement following it in the program unit (that is, one at the same IF-level). Each block IF statement may also have zero or one corresponding ELSE statement and any number of corresponding ELSE IF statements following it in the program unit (at the same IF-level).

2225 EXPRESSION IN ELSE IF STMT. IS NOT TYPE LOGICAL

The expression *e* in the ELSE IF (*e*) THEN statement must be a logical expression. The ELSE IF statement is ignored.

2226 A DO-LOOP AND AN IF-BLOCK ARE INCORRECTLY NESTED

If a DO statement appears within an IF-block, ELSE IF-block, or ELSE-block, the range of the DO-loop must be contained entirely within that block. Similarly, if a block IF statement (that is, IF (*e*) THEN) appears within the range of a DO-loop, the corresponding END IF statement must also appear within the range of that DO-loop.

2227 ELSE OR ELSE IF STATEMENT MAY NOT FOLLOW ELSE STATEMENT

Once an ELSE statement has appeared at a given IF-level, an END IF statement must appear before the next ELSE or ELSE IF statement at the same IF-level. The ELSE or ELSE IF statement is ignored.

2228 BLOCK IF STATEMENT REQUIRES MATCHING END IF STATEMENT

Each block IF statement (that is, IF (*e*) THEN) must have exactly one corresponding END IF statement following it in the program unit. An implied END IF statement is inserted at the end of the program unit to match the block IF.

2229 STATEMENT CANNOT TERMINATE A DO-LOOP

The terminal statement of a DO-loop must not be a nonexecutable statement (except FORMAT) or one of the following executable statements: an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement.

2301 FORMAT CODE *xx* IS INCORRECT

The indicated letter is not a valid format field identifier. The letter and all following characters are deleted up to the first following delimiter other than a period.

2302 FORMAT CODE IS NOT FOLLOWED BY AN INTEGER

The indicated format code letter is not followed by a field width designator. The format code letter is deleted.

2303 FORMAT CODE INTEGER COMPONENT IS TOO LARGE

The indicated integer is greater than 255. The value 255 is used instead of the large number.

2304 FORMAT CODE *xx* FRACTIONAL PART IS MISSING

Either a format code which requires a "d" subfield (such as *Ew.d* or *Fw.d*) does not have the "d" subfield, or a format code for which the "d" subfield is optional (such as *Iw [.d]*) has the period delimiter, but no value is specified for *d*. The value 0 is used for the missing subfield.

2305 FORMAT CODE *xx* HAS ZERO FIELD WIDTH

The field width for the indicated format code has been specified as zero. The format code is deleted.

2306 FORMAT HAS EMPTY LITERAL FIELD

The format statement contains a literal field containing zero characters. The empty literal field is deleted.

2307 REPEAT COUNT IS ZERO FOR *xx*

The repeat count for the indicated format code or parenthesis is zero. The format is processed as though the zero repeat count had not been specified.

2308 SCALE FACTOR IS NOT PRECEDED BY AN INTEGER

A "P" field is not preceded by an integer. The value 0 is used.

2309 FORMAT PARENTHESIS NESTING IS TOO DEEP

Parentheses are nested more than five deep, including the outermost pair which begin and end the statement. The format is translated as though no error had occurred, but group repetition will not be done properly by the execution time format scanner for groups which are too deep in the parenthesis nest.

2310 HOLLERITH FIELD AMBIGUITY - FORMAT DELETED

Because of the nature of its lexical analysis algorithm, ASCII FORTRAN is sometimes unable to correctly identify a literal field which is not preceded by the required delimiter (comma, slash, or left parenthesis). A preceding message has identified the point at which the delimiter is required. The FORMAT statement is replaced by an empty FORMAT, which forces list-directed input and output.

2401 ARITHMETIC IF EXPRESSION IS OF WRONG TYPE

The parenthesized expression in the IF statement is not logical, so the IF is assumed to be arithmetic. However, the expression is not INTEGER, REAL, or DOUBLE PRECISION, which are the only types for which the required relations to zero are defined. The statement is ignored.

2501 FILE REFERENCE NUMBER *xx* IS INCORRECT

In a DEFINE FILE statement, the file reference number is not an integer, or has already appeared as the file reference number of a previous DEFINE FILE statement. The second DEFINE FILE statement is deleted. This message is also generated if the file reference number of a BACKSPACE, REWIND, ENDFILE, FIND, READ, or WRITE statement is not an integer expression or a character variable, character array name, or character array element where these are permitted. The statement is deleted.

2502 RELATIVE RECORD NUMBER *xx* IS INCORRECT

The record number specification of a direct access I/O statement is not an integer expression. The statement is deleted.

2503 INPUT LIST MEMBER *xx* IS AN ACTIVE DO VARIABLE

The indicated variable appears in an input list. It is the index variable of an active DO or implied-DO. The input list is ignored.

2504 NAMELIST REFERENCE *xx* IS INCORRECT

The item encountered where the NAMELIST name was expected is not a valid name. The NAMELIST statement is deleted.

2505 ASSOCIATED VARIABLE *xx* IS INCORRECT

The associated variable of this DEFINE FILE or OPEN either is not an integer scalar or was used as the associated variable of a prior DEFINE FILE or OPEN statement. This DEFINE FILE or OPEN statement is deleted.

2506 FILE FORMAT SPECIFICATION *xx* IS INCORRECT

The DEFINE FILE format specification is not one of the letters L, E, U, M, F, or V. The statement is deleted.

2507 NUMBER OF RECORDS FIELD *xx* IS INCORRECT

The *number-of-records* field of the DEFINE FILE statement is not an integer constant. The statement is deleted.

2508 SIZE OF RECORD FIELD *xx* IS INCORRECT

The *size-of-records* field of the DEFINE FILE statement requires a positive integer constant less than 262,144, which was not found. The statement is deleted.

2509 CHARACTER NUMBER *xx* IS INCORRECT

The record size specification of the ENCODE or DECODE statement is not an integer constant, variable, or array element, or is negative. The statement is deleted.

2510 STARTING LOCATION *xx* IS INCORRECT

The location to which the ENCODE or DECODE data transfer is to be made is incorrectly specified. It should be an array, array element, or variable. The statement is deleted.

2511 CHARACTERS PROCESSED NAME *xx* IS INCORRECT

The entity specified to receive the number of characters processed by the ENCODE or DECODE is not an integer variable or array element. The statement is deleted.

2512 FORMAT REFERENCE *xx* IS INCORRECT

The FORMAT specification must be a FORMAT statement number, an integer variable containing a FORMAT statement label, or an array or character expression containing a run-time FORMAT. This message is generated if the FORMAT specification is a statement label but not a FORMAT statement label or is a variable but not of type integer or character. This message is generated during the optimization process if it is determined that specified integer variable will not have been assigned a FORMAT statement label in an ASSIGN statement when the referencing statement is executed. In any case, the statement is deleted.

2513 DIRECT ACCESS OR INTERNAL FILE I/O IS LIST OR NAMELIST DIRECTED

The FORMAT specification in this direct access or internal file I/O statement is a NAMELIST name or an asterisk. Only formatted and unformatted I/O are permitted for direct access files, and only formatted sequential I/O is permitted for internal files.

2514 REDUNDANT 'ERR=' OR 'END='

A second "ERR=" or "END=" clause has been encountered within the same statement. The repeated clause is ignored.

2515 'END=' RETURN IS USED INCORRECTLY

An "END=" clause has been specified in a direct access READ or WRITE statement. The "END=" clause is ignored.

2516 INPUT LIST MEMBER *xx* IS AN EXPRESSION OR CONSTANT

The members of an input list must be variables, elements, or arrays. The input list is ignored.

2517 I/O LIST PRESENT WITH NAMELIST DIRECTED I/O

An I/O list has been specified on a READ, WRITE, PRINT, or PUNCH statement which has a NAMELIST name instead of a FORMAT specification. The list is ignored.

2520 I/O LIST MEMBER *xx* IS INCORRECT

The indicated item is not permitted to appear in an I/O list. The I/O list is ignored.

2521 DEFINE FILE OPTION *xx* IS INCORRECT

The indicated option is not valid for a sequential DEFINE FILE. The statement is deleted.

2522 FILE REFERENCE NUMBER IS OUT OF RANGE

The file reference number is negative or zero or greater than 262,143. The statement is deleted.

2523 INTERNAL FILE REQUEST IS INCORRECT

Internal file I/O may be formatted and sequential only.

2524 IOSTAT FIELD IS INCORRECT

An IOSTAT= specification may only be an integer variable or an integer array element.

2525 KEYWORD *xx* IS USED INCORRECTLY

A keyword used in an OPEN, CLOSE, or INQUIRE statement conflicts with another keyword, or is not permitted in this statement.

2526 SPECIFICATION FOR *xx* IS INCORRECT

The specification for this keyword is of the wrong type or is an expression where this is not permitted.

2527 ONLY UNIT, IOSTAT, AND ERR MAY BE PRESENT WITH REREAD

2528 KEYWORD *xx* NOT PERMITTED IN CLOSE STATEMENT2529 KEYWORD *xx* PERMITTED ONLY IN INQUIRE STATEMENT2531 KEYWORD *xx* NOT PERMITTED IN THIS STATEMENT

2532 UNIT OR FILE SPECIFICATION MUST BE PRESENT

2533 UNIT SPECIFICATION MUST BE PRESENT

2602 VARIABLE *xx* MAY BE USED BEFORE SET TO A VALUE

The analysis of program flow has determined that there exist possible paths in the routine along which the value of the indicated variable may be referenced before any value has been assigned. It is the programmer's responsibility to ensure that the variable is, in fact, initialized on all possible paths, or that the paths along which the incorrect reference occurs cannot be executed.

This warning message will also be given when a character variable is only partially (that is, a substring of the character variable) defined by an assignment statement before a use of all byte positions, of the character variable. It is the programmer's responsibility to ensure that all byte positions of the character variable have been defined before the use of the entire character variable.

2603 ARITHMETIC ERROR IN DO LOOP CALCULATION

This error may be issued in either of two cases:

1. The evaluation of a constant-valued expression while calculating the iteration count has caused an arithmetic fault; or
2. The evaluation of a constant-valued expression, while performing the strength reduction phase of global optimization, has caused an arithmetic fault.

2604 TRANSFER OF CONTROL INTO DO LOOP INHIBITS OPTIMIZATION

The branching structure of a DO-loop is too complex to be analyzed by the optimization phase. The code will be correct, but not optimal. The complexity results from branches out of and into the DO-loop; most probably the branches are from the extended range of the DO-loop. The user should consider simplifying the branching structure of the DO-loop in order to obtain optimal code from this compiler.

2605 MAXIMUM NUMBER OF REDUCIBLE EXPRESSIONS IN DO LOOP

The maximum number of reducible expressions for a DO-loop has been encountered. Any further expressions in the loop will not be considered for optimization.

2606 VARIABLE *xx* IS REFERENCED BUT IS NEVER ASSIGNED A VALUE

A variable name is referred to in the program but nowhere in the program is the variable ever assigned any value. A variable may be assigned a value in a number of ways including an assignment statement, a DATA statement, use of the variable as a subprogram argument and many others. Ordinarily, reference to such a variable with no assigned value will refer to a zero value. However, the value referred to may be unknown depending upon the options used on the compilation and collection of the program.

2607 VARIABLE *xx* APPEARS IN A DECLARATION BUT IS NEVER REFERENCED

The variable is specified in a type or DIMENSION statement but neither it nor any overlays is ever used in an executable statement. The program should be checked for misspellings, and elimination of the unnecessary variable should be considered.

2608 DUMMY ARGUMENT *xx* IS NEVER REFERENCED

The dummy argument appears in the argument list of a FUNCTION, SUBROUTINE or ENTRY statement but is never used in an executable statement. The program should be checked for misspellings, and elimination of the unnecessary argument should be considered.

3100 STMT FUNCTION *xx* TYPED AS CHARACTER*(*)

The FORTRAN 77 standard requires a character statement function to have a length specification that is a constant integer expression. CHARACTER*(*) is not allowed.

3101 STMT FUNCTION *xx* USED AS TARGET OF ASSIGN. STMT

The FORTRAN 77 standard requires a statement function reference to appear in an expression. It does not allow a statement function reference to appear on the left-hand side of an assignment statement (at the outer level). ASCII FORTRAN allows this, and does not perform any type conversion (to convert the expanded statement function expression to the type of the statement function).

3102 *xx* USED AS STMT FUNCTION ACTUAL ARG

The FORTRAN 77 standard requires a statement function actual argument to be an expression. It cannot be a statement label, an array name, or a function name.

3103 CHAR. FUNCTION ENTRY POINTS HAVE DIFF. LENGTHS

The FORTRAN 77 standard states that in a character function, all entry points must be typed with the same length specification; that is, all entry point names must have the same constant length (for example, CHARACTER*2), or all entry point names must be typed CHARACTER*(***).

3104 MULTIPLE ASSIGNMENT STATEMENT

The FORTRAN 77 standard allows only one target variable in an assignment statement. ASCII FORTRAN allows the source item (on the right-hand side of the equal sign) to be stored to more than one target item (on the left-hand side of the equal sign, separated by commas).

3105 NON-CHARACTER ITEM SET TO CHAR. CONSTANT

The FORTRAN 77 standard does not allow a noncharacter item to be the target (left-hand side of the equal sign) of an assignment statement, where the source item (right-hand side of the equal sign) is a character constant, since there is no character to noncharacter conversion allowed. The standard has arithmetic assignment statements and character assignment statements, with no mixing allowed. ASCII FORTRAN allows this, with no conversion performed on the character constant (that is, it is simply stored to the target variable).

3106 DIMENSION FOLLOWS LENGTH FOR ARRAY *xx*

The FORTRAN 77 standard states that the local length specification (if any) for a character type statement must follow the dimension information (if any). For example, the standard requires CHARACTER A(2)*3, while ASCII FORTRAN allows that syntax or CHARACTER A*3(2).

3107 DATA INIT. IN SPEC. STMT. FOR *xx*

The FORTRAN 77 standard allows data initialization only in the DATA statement. ASCII FORTRAN also allows it in explicit type and DIMENSION statements.

3108 LENGTH SPEC. **n* APPEARS FOR NON-CHAR. ITEM

The FORTRAN 77 standard allows a length specification of **n* (where *n* is an integer constant) or *(*e*) (where *e* is an integer constant expression) only for type character, in the explicit type, FUNCTION, and IMPLICIT statements. For example, the forms REAL*4, REAL*8, and COMPLEX*8 are not allowed in the standard, but CHARACTER*2, CHARACTER*(2+3), and CHARACTER*(***) are allowed.

3109 COMPLEX*16 DATA TYPE IS UNDEFINED

The FORTRAN 77 standard defines the complex data type to be single precision (that is, composed of real and imaginary parts that are both single precision real). ASCII FORTRAN allows single precision complex (COMPLEX*8) and double precision complex (COMPLEX*16).

3110 COMPLEX*16 CONSTANT ENCOUNTERED

The FORTRAN 77 standard has no double precision complex data type (only single precision complex). Therefore, a COMPLEX*16 constant is not allowed.

3111 FORMAT EDIT DESCRIPTOR *xx* IS UNDEFINED

The repeatable edit descriptors *Jw*, *Ow*, *Rw*, and *Ew.dDe* are not defined in the FORTRAN 77 standard. The nonrepeatable edit descriptors *+S*, *-S*, and *-wX* are not defined in the standard. These edit codes are allowed in ASCII FORTRAN formats.

3112 *xx* CLAUSE ON I/O STATEMENT IS UNDEFINED

The listed clause is not allowed on the current input/output statement (OPEN, CLOSE, or INQUIRE) in the FORTRAN 77 standard.

3113 END CLAUSE IN WRITE STATEMENT

The FORTRAN 77 standard does not allow an *END=s* clause on a WRITE statement.

3114 *u'r* SYNTAX USED FOR DIRECT ACCESS READ OR WRITE

The FORTRAN 77 standard does not allow an apostrophe (') to designate direct access I/O, as does ASCII FORTRAN. The standard has separate *UNIT=u* and *REC=r* clauses for the direct access READ and WRITE statements.

3115 NON-CHAR. ARRAY *xx* USED AS FORMAT SPECIFIER

The FORTRAN 77 standard allows the following for a format specifier in I/O statements: statement label, integer variable, character expression, asterisk, or character array name. ASCII FORTRAN allows noncharacter array names, in addition to the standard list.

3116 NAMELIST NAME *xx* USED IN I/O STATEMENT

The FORTRAN 77 standard does not allow namelist I/O. The current statement is an I/O statement with one of the following formats:

```
READ(u,n,...)  
READ n  
WRITE(u,n,...)  
PRINT n  
PUNCH n
```

where *n* is a namelist name (declared in a NAMELIST statement).

3117 NON-INTEGERS SUBSCRIPT USED FOR ARRAY *xx*

The FORTRAN 77 standard requires a subscript in an array reference to be an integer expression. ASCII FORTRAN allows real or double precision expressions, as well as integer expressions.

3118 CHARACTER*(*) ITEM *xx* APPEARS IN CONCAT.

The FORTRAN 77 standard allows a CHARACTER*(*) item (dummy scalar, dummy array element, or character function entry point) to appear as a concatenation operand only in a character assignment statement, and even then the concatenation may not appear in an argument (to either a function or statement function). ASCII FORTRAN allows a CHARACTER*(*) item to appear anywhere a character expression is allowed.

3121 CHARACTER LENGTH SPEC FOLLOWS FUNCTION NAME

The FORTRAN 77 standard allows the syntax in the following example for a character function specification:

CHARACTER*5 FUNCTION C(arg)

ASCII FORTRAN also allows:

CHARACTER FUNCTION C*5(arg)

3122 PARAMETER STATEMENT OCCURS AMONG EXECUTABLE STMTS

In the FORTRAN 77 standard, PARAMETER statements are only allowed to be mixed with IMPLICIT statements and specification statements. They can not occur after any DATA statements, statement function definitions, or executable statements.

3123 SPEC. STMT OCCURS AFTER DATA OR STMT FUNCTION

The FORTRAN 77 standard forces all specification statements to occur before any DATA statements or statement function definitions are encountered.

3124 INTERNAL SUBPROGRAMS ARE NON-STANDARD

The FORTRAN 77 standard does not have the ASCII FORTRAN concept of internal subprograms, where a SUBROUTINE or FUNCTION statement occurring inside of another program unit with no intervening END statement causes the subprogram to be local to the containing external subprogram and to share its declarations.

3125 STATEMENT NOT ALLOWED IN BLOCKDATA SUBPROGRAM

The FORTRAN 77 standard does not allow the INTRINSIC or EXTERNAL statements to occur within a blockdata subprogram.

3126 STATEMENT IS NON-STANDARD

ASCII FORTRAN has the following statements which are not in the FORTRAN 77 standard: FIND, PUNCH, ENCODE, DECODE, BANK, DEFINE, NAMELIST, DEBUG, AT, DEFINE FILE, START EDIT, STOP EDIT, DELETE, INCLUDE, DISPLAY, TRACE, COMPILER.

3127 OVER 19 CONTINUATION LINES ENCOUNTERED

The FORTRAN 77 standard allows up to 19 continuation lines. ASCII FORTRAN allows as many as necessary as long as the number of significant characters is under approximately 1320.

3128 *xx* EQUIVALENCED TO A *xx* ITEM

The FORTRAN 77 standard does not allow character type items to be equivalenced to noncharacter type items.

3129 *xx* COMMON BLOCK CONTAINS CHARACTER AND NONCHARACTER ITEMS

The FORTRAN 77 standard does not allow character type items to be in the same common block as noncharacter type items.

3130 ARRAY *xx* EQUIVALENCED WITH WRONG NUMBER OF SUBSCRIPTS

The FORTRAN 77 standard does not allow an array to appear in equivalence with a number of subscripts that differs from the number of dimensions specified for the array. ASCII FORTRAN allows one subscript to appear on arrays in equivalence even if the array was dimensioned with more than one dimension.

3131 PARENTHESIS MISSING ON FUNCTION STATEMENT

ASCII FORTRAN allows a FUNCTION statement without parenthesis if the function has no arguments. The FORTRAN 77 standard requires them.

3132 SLASHES *xx* BRACKET DUMMY ARGUMENT

For compatibility reasons, ASCII FORTRAN allows the user to bracket a dummy argument with slashes:

```
SUBROUTINE SUBX(a,/b/)
```

The FORTRAN 77 standard does not allow this.

3133 *xx* USED INSTEAD OF ASTERISK TO INDICATE A LABEL

ASCII FORTRAN allows a currency symbol (\$) to indicate a label in a dummy argument list, and a currency symbol or ampersand (&) to precede a label when passing a label to a subprogram. The FORTRAN 77 standard uses an asterisk (*) only.

3134 FUNCTION SUBPROGRAMS SHOULD NOT HAVE LABEL ARGUMENTS

The FORTRAN 77 standard only allows label arguments to be passed to and accepted by SUBROUTINES. ASCII FORTRAN also allows them with FUNCTION subprograms.

3135 *xx* USED IN EXTERNAL STATEMENT

The FORTRAN 77 standard only allows a list of routine names on an EXTERNAL statement. ASCII FORTRAN allows the names to be preceded by & or * (& is ignored; * means it is a FORTRAN V subprogram). Also, ASCII FORTRAN allows the routine name to be followed by (*opt*), where *opt* is ACOB, PL1, or FOR to indicate the language of the routine (FOR is FORTRAN V).

3136 FUNCTION SUBPROGRAM HAS AN ALTERNATE RETURN

The FORTRAN 77 standard does not allow labels to be passed to functions and therefore does not allow alternate returns. ASCII FORTRAN allows both.

3137 MAIN PROGRAM HAS A RETURN STATEMENT

ASCII FORTRAN treats a RETURN statement in a main program as a STOP statement. The FORTRAN 77 standard does not allow a RETURN statement in a main program.

3138 PARAMETER STATEMENT HAS MISSING PARENTHESIS

According to the FORTRAN 77 standard, a PARAMETER statement needs opening and closing parentheses:

```
PARAMETER (I=2,J=3,CHAR2='2')
```

ASCII FORTRAN also allows the form:

```
PARAMETER I=2,J=3,CHAR2='2'
```

3139 INTRINSIC FUNCTION *xx* USED IN CONSTANT-EXPRESSION

The FORTRAN 77 standard states that only constant-valued expressions can be used in certain places such as the length specification in a PARAMETER statement. The standard defines constant-valued expressions as only having constants, other parameter constants, and simple operators. No intrinsic functions such as SIN, REAL, and CHAR may be used, and only simple integer exponentiation may be used. ASCII FORTRAN allows the intrinsic functions and general exponentiation to be used in any place where constant expressions are required (except for DIMENSION declarators). However, these are not available for use in constant-valued expressions if the compiler was generated such that the intrinsic functions are not available at compile time. This is rarely done, since the compiler as released is not generated in this manner. ASCII FORTRAN allows:

```
PARAMETER (SINPI1 = 1.0 + SIN(3.1416))
```

3140 LABEL *xx* ENCOUNTERED IN INITIALIZATION LIST

The FORTRAN 77 standard does not allow items to be initialized to label values in DATA statements, but ASCII FORTRAN does:

```
DATA I/&10/
```

3141 DATA or FORMAT STATEMENT USED TO TERMINATE DO-LOOP

ASCII FORTRAN allows a DATA or FORMAT statement to be the terminator of a DO-loop; the FORTRAN 77 standard does not.

3142 OCTAL CONSTANT ENCOUNTERED IN INITIALIZATION LIST

ASCII FORTRAN allows octal values to be put into variables in DATA statements, and the FORTRAN 77 standard does not recognize the letter "o" for octal. For example:

```
DATA I/o040040/
```

3143 FIELDATA CONSTANT ENCOUNTERED IN INITIALIZATION LIST

ASCII FORTRAN allows Fieldata values to be put into variables in DATA statements; the FORTRAN 77 standard has no Fieldata data type. For example:

```
DATA i/'abcdef'F/
```

3144 BLANK COMMON VARIABLE *xx* INITIALIZED

The FORTRAN 77 standard prohibits blank COMMON from being initialized with DATA statements. ASCII FORTRAN allows it.

3145 COMMON VARIABLE *xx* INITIALIZED IN NON-BLOCKDATA SUBPROGRAM

The FORTRAN 77 standard allows COMMON to be initialized only inside of a BLOCK DATA subprogram. ASCII FORTRAN allows DATA statements on COMMON in any kind of program unit.

3146 NON-CHARACTER ITEM INITIALIZED TO A CHARACTER VALUE

The FORTRAN 77 standard does not allow noncharacter items to be initialized to character values in DATA statements. ASCII FORTRAN does allow this.

3147 DIGIT STRING *xx* OVER 5 DIGITS LONG

The FORTRAN 77 standard only allows up to a 5-digit string on a PAUSE or STOP statement. ASCII FORTRAN allows more.

3148 HOLLERITH LITERALS ARE ONLY ALLOWED IN FORMAT STATEMENTS

The FORTRAN 77 standard does not recognize Hollerith anywhere but in a FORMAT statement. ASCII FORTRAN allows Hollerith wherever a character constant may occur. Note, however, that when passed as arguments to a subprogram, Hollerith should *not* be passed if the dummy argument in the receiving program is type CHARACTER. If Hollerith is passed as an argument, the dummy argument should be anything but type character. (This seemingly strange situation is spelled out in the FORTRAN 77 standard, Appendix C, paragraph C7. It is done this way to allow existing programs using no character data type to continue to operate correctly under the new standard.)

3149 PARAMETER CONSTANT *xx* USED AS PART OF COMPLEX CONSTANT

The FORTRAN 77 standard does not allow the use of PARAMETER constants as the REAL or IMAGINARY portions of a complex constant. ASCII FORTRAN does allow this. For example:

```
COMPLEX c  
PARAMETER (p1=2.,p2=3.)  
c = (p1,p2)
```

3150 '\$' USED AS A CHARACTER IN THE NAME *xx*

Only the letters A-Z and the digits may be used in a name under the FORTRAN 77 standard. ASCII FORTRAN also allows "\$" to be used, though it is discouraged since accidental conflicts with the run-time library entry points may occur if subprograms have \$ in their names.

3151 '&' USED AS CONCATENATION OPERATOR

The concatenation operator is the double slash (//). ASCII FORTRAN also allows the use of the ampersand (&).

```
CHARACTER*80 c80,c22*22,c2*2
c80 = c22//c2
c80 = c22&c2
```

3152 COMPUTED GOTO HAS EMPTY LABEL POSITION(S)

FORTRAN 77 does not allow missing positions in a computed GOTO. ASCII FORTRAN does allow void positions. If a void position is selected, the next statement in sequence is executed.

```
GO TO (10,20,,40) I+2
```

3153 INTEGER VARIABLE *xx* USED FOR A LABEL

FORTRAN 77 requires labels in the lists that arithmetic IFs and computed GOTOs have. ASCII FORTRAN also allows integer variables, which must have received the value of some label via an ASSIGN or DATA statement.

```
ASSIGN 10 TO ILAB
GOTO (20,30,ILAB,40) I+3
20 IF(J+2) 30,ILAB,40
```

3154 ARITHMETIC IF HAS EMPTY OR MISSING LABEL POSITIONS

FORTRAN 77 does not allow empty or missing positions in an arithmetic IF. ASCII FORTRAN allows this, and if that position is selected, the next statement is executed.

```
I=2
IF(I) 10,20
```

In this case, the next statement is executed.

3155 *xx* SHOULD BE IN AN *xx* STATEMENT

The FORTRAN 77 standard says that any subprogram name which is passed as an argument must be either in an EXTERNAL statement (if it is a user-supplied subprogram), or in an INTRINSIC statement (if it is a FORTRAN intrinsic function). ASCII FORTRAN also allows the names to be passed if they were used previously in source as a valid subprogram reference.

```
X = SIN(A)
Y = MYPROG(B)
CALL SUB2(SIN,MYPROG)
```

The previous example is nonstandard, though ASCII FORTRAN allows it to go through with no error.

3156 **xx INLINE COMMENTS ENCOUNTERED IN xx**

Inline comments are nonstandard in FORTRAN 77. For example:

```
      x = 1.5     @ comments here
```

3157 **FUNCTION xx IS NOT A STANDARD INTRINSIC**

The function referred to is an intrinsic function in ASCII FORTRAN, but is not included in FORTRAN 77.

3158 **ARGUMENT TYPE IS INCORRECT FOR FUNCTION xx**

ASCII FORTRAN allows any intrinsic function to be used as a generic name of a function; for example, DSIN(real) will call SIN automatically. This capability is limited to a small number of generic functions in FORTRAN 77.

6301 **'BANK' STATEMENT IGNORED IN 'CHECKOUT' MODE**

A user module compiled and executed with the checkout option (@FTN,C) must be self-contained. Since it is not collected into an absolute, the BANK statement has no meaning and is ignored.

6302 **'COMPILER' STATEMENT IGNORED IN 'CHECKOUT' MODE**

The BANKED= options of the COMPILER statement are not allowed in checkout mode since the user's program cannot be banked by the user. Note that the LINK=IBJ\$ and the DATA= options are allowed, however.

Appendix E. Conversion Table

Table E-1 (on the next page) shows if a specific data type can be converted to another desired data type. If conversion is possible, a brief description of the method of conversion is given.

Table E-1. Conversion Methods for Arithmetic Data

Desired Type	Present Type of Expression							Statement Number
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	COMPLEX *16	Octal	Logical or Character (see note)	
INTEGER	Store.	Fix. Store.	Fix. Store.	Fix. Store real part. Ignore imaginary part.	Fix. Store real part. Ignore imaginary part.			Store.
REAL	Float. Store.	Store.	Contract to single. Store.	Store real part. Ignore imaginary part.	Truncate real part to single. Store. Ignore imaginary part.	Not possible, except in DATA stmt. (See 6.8.1 for details.)	Not possible, except in DATA stmt. (See 6.8.1 for details.)	Not possible.
DOUBLE PRECISION	Double float. Store.	Extend to double. Store.	Store.	Extend real part to double. Store. Ignore imaginary part.	Store real part. Ignore imaginary part.	6.8.1 for details.	6.8.1 for details.	
COMPLEX	Float. Store to real part. Store 0 as imaginary part.	Store to real part. Store 0 as imaginary part.	Contract to single. Store to real part. Store 0 as imaginary part.	Store.	Contract both parts to single. Store.			
COMPLEX *16	Double Float. Store to real part. Store 0 as imaginary part.	Extend to double. Store to real part. Store 0 as imaginary part.	Store to real part. Store 0 as imaginary part.	Extend both parts to double. Store.	Store.	Not possible, except in DATA stmt.	Not possible, except in DATA stmt.	Not possible.
Character or Logical	Not possible.						Truncate or blank fill, if necessary, for character.	

NOTE: A character literal or Hollerith string may be assigned to any type of data item. See 3.3 for details.

Appendix F. Tables of FORTRAN Statements

Tables F-1 and F-2 specify whether a FORTRAN statement is executable or nonexecutable.

Table F-1. Nonexecutable Statements

General Category	Statement
Specification statements	DIMENSION COMMON EQUIVALENCE BANK EXTERNAL NAMelist PARAMETER INTRINSIC IMPLICIT Explicit Type statements: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER SAVE
Data initialization statement	DATA
Format statement	FORMAT
Function defining statement	Statement function definition (DEFINE)
Program unit headings	PROGRAM FUNCTION SUBROUTINE ENTRY BLOCK DATA
Program control statements	INCLUDE DELETE EDIT (START EDIT, STOP EDIT) COMPILER
Debug facility	DEBUG AT

Table F-2. Executable Statements

General Category	Statement
Assignment statements	Arithmetic assignment statement Logical assignment statement Character assignment statement ASSIGN
Control statements	Unconditional, assigned, and computed GO TO statements Arithmetic and logical IF statements Blocking statements (block IF, ELSE IF, ELSE, and END IF) CALL CONTINUE RETURN STOP PAUSE DO END
I/O statements	READ WRITE PRINT PUNCH REWIND BACKSPACE ENDFILE DEFINE FILE FIND ENCODE DECODE OPEN CLOSE INQUIRE
Debug facility	TRACE ON TRACE OFF DISPLAY

Appendix G. ASCII FORTRAN Input/Output Guide

G.1. General

ASCII FORTRAN performs run-time input/output using the Processor Common I/O Modules. Processor Common I/O consists of a number of common bank modules which access the various file types and a number of processor interface common bank modules for the various ASCII processors such as COBOL, FORTRAN, and PL/I.

Code compiled by ASCII FORTRAN references the ASCII FORTRAN I/O processor interface module which references the required Processor I/O Common Bank Modules. The Processor Common I/O System (PCIOS) is used by the ASCII FORTRAN run-time system for I/O compatibility between the various language processors (see the Processor Common Input/Output System (PCIOS) Interface Description, UP-8478 (see Preface)). All ASCII FORTRAN SDF and ANSI files are accessed via PCIOS. The specialized ASCII FORTRAN interface module handles all formatting and conversions, with the actual I/O being done in PCIOS modules. However, because of a PCIOS restriction, I/O processing cannot be done to word-addressable mass storage. The ASCII FORTRAN interface module also handles all symbiont I/O (PRINT, PUNCH, READ).

The ASCII FORTRAN programmer may access the following file formats:

- SDF (System Data Format)
- ANSI (American National Standard Institute Magnetic Tape Interchange Format)
- ASCII symbiont

G.2. System Data Format (SDF) File

An SDF file produced by ASCII FORTRAN is explained in this subsection.

G.2.1. SDF File Description

An SDF file produced by ASCII FORTRAN contains a label record, data records or record segments, an end-of-file record, and possibly bypass records.

Each record or record segment has a control word in front of it. This control word has the form:

T1	T2	S5	S6
<i>l</i>	<i>p</i>	<i>f</i>	<i>s</i>

where:

l is the length of the data record or record segment if bit 35 is set to zero. A data record or record segment length must be in the range zero to 2,047 words. If bit 35 is not zero, the value of *l* may be:

<u>Value</u>	<u>Meaning</u>
05400	End-of-reel
05033	Label image
05100	Continuation
07700	End-of-file image
040 <i>x</i>	Bypass images where <i>x</i> is the length of the bypass image with a range $0 \leq x \leq 63$

p is the *l* field from the previous record or record segment. This field is used in backspacing.

f is:

<i>n</i>	For unformatted records this field contains a count <i>n</i> of the number of bits used in the last word of the record.
077	Dummy record. Dummy records are produced when a direct access file is skeletonized.

s is the record segment indicator:

<u>Value</u>	<u>Meaning</u>
00	Record not segmented
01	First segment of record
02	Last segment of record
03	Not first or last segment of the record

SDF records are segmented to conserve main storage space when processing large unformatted records.

The direct access SDF has the same format as a sequential SDF file except all records in the file are the same length. When a direct access file is skeletonized, dummy records are written in the file.

G.2.1.1. SDF Labels

The SDF label contains the file's maximum record size, the block size the file was written with, the record segment size, file type, and other information pertinent to processing this file. The label record is the first record of the file.

The format of a PCIOS SDF label record is:

0	050	033	035	<i>ASCII flag</i>
1	<i>file</i>			
2	<i>name</i>			
3	<i>type</i>	<i>orig</i>	<i>level</i>	<i>recovery offset</i>
4	<i>block size</i>		<i>record size</i>	
5	<i>date</i>			
6	<i>address of next file</i>			
7	<i>largest record key allowed</i>			
8	<i>largest record key written</i>			
9	<i>rcd\1\offset</i>		<i>segment size</i>	
10	<i>ffc</i>	<i>skel flag</i>	0	<i>record size in characters</i>
11	<i>address of previous file</i>			
12	0			
13	0			
14	.			
15	.			
16	.			
17	.			
18	.			
19	.			
20	.			
21	.			
22	.			
23	.			
24	.			
25	.			
26	.			
27	.			

(Words 14 through 27 are written as they appear in the label area at open time.)

Word 0 is the label control word in which a Fielddata X (035) identifies this SDF file as having been made by the Processor Common Input/Output System (PCIOS). The ASCII flag has a value of 1 for ASCII files.

Words 1 and 2 contain the internal file name used when the file was created.

In Word 3:

type indicates the type of SDF file:

01 = sequential

02 = direct

orig indicates the originator:

01 = ASCII FORTRAN pre-level 9R1

02 = ASCII PL/I

03 = ASCII FORTRAN

04 = ASCII COBOL

level indicates that the C2DSDF/C2SSDF file was created or extended by PCIOS level 4R1 or higher.

recovery offset is valid only if level is set; it is set by C2DSDF/C2SSDF and holds the word offset of record $n + 1$ in the last file block. C2SSDF uses this for output extend recovery in the event of system failure during an extend operation.

In Word 4:

block size specifies the block size (in words) of the file.

record size specifies the record size (in words) of the file.

Word 5 contains the date when the file was created.

Word 6 contains the mass storage address of the next file for sequential stacked files on mass storage. This field will be zero for SDF direct files.

Word 7 indicates the maximum relative record number which is allowed for a direct SDF file. This word is zero for sequential files.

Word 8 specifies the highest record number actually written in this direct SDF file. This word is zero for sequential files.

In Word 9:

rcd\1\offset indicates the word offset for relative record number 1.

segment size indicates the segment size (in words) used when creating this file.

In Word 10:

ffc is the FORTRAN record format control code.

skel flag is a skeletonization flag for direct SDF files.

*record size
in characters* is the record size in characters of the file.

Word 11 contains the address of the previous file if files are stacked on mass storage. For the first file, this word is -0 since there is no previous file.

Presently, words 12 and 13 are not used.

Words 14 through 27 will be written as they appear in the label area (PTLFMA) when the open output was called.

G.2.1.2. SDF Data Records/Record Segments

SDF data records can be character data or word oriented. The length of an SDF data record is maintained in the file as words. An SDF record can be any length but they will be automatically segmented when the record is written if they exceed the segment size for that particular file.

The segment size for a particular file is specified in the File Control Table (FCT) for that file. The segment size for an SDF file is specified by the user in the SEG clause of the OPEN statement or in the *ss* field of the sequential DEFINE FILE statement. If no OPEN or DEFINE FILE statement is specified and the file is an input file, the segment size field of the SDF label is used.

If the file is an output file and no OPEN or DEFINE FILE statement was specified, the default segment size specified in the Storage Control Table (SCT) is used (see G.7).

In order to minimize I/O overhead when processing sequential files, a segment size greater than or equal to the record size should be specified in the OPEN or DEFINE FILE statement.

The maximum record size for the SDF file pertains to records other than unformatted records, since these other records are not processed by segments (although they are segmented when they are written in the file) like unformatted records are. Records other than unformatted records are processed by the FORTRAN editing routines as complete records. To ensure there is space for the complete record in the intermediate record area, the maximum size that may be encountered must be known. The maximum record size for a particular file is maintained in the FCT for that file. The maximum record size for an SDF sequential file is specified by the user with the MRECL clause of the OPEN statement or in the *rs* field of the DEFINE FILE statement. If no OPEN or DEFINE FILE statement is specified and file is an input file, the maximum record size field of the SDF label is used. The default maximum record size specified in the SCT is used for output files if no OPEN or DEFINE FILE statement was specified (see G.7).

G.2.1.3. SDF Block Size

The block size for an SDF sequential data file is specified by the BLOCK clause of the OPEN statement or by the *bs* field of the sequential DEFINE FILE statement. If not specified, the default size is 224 words. The block size is independent of record size since records span the block when necessary. Record size can be larger than block size. When choosing a block size, the user should be aware of the following:

- The block size field *bs* (or *bsize*) specifies the number of words in the block. In order to minimize I/O overhead, block size should be larger than segment size.
- Tape files must be processed with a block size at least as large as the block size they were created with.
- Mass storage files must be processed with the exact size they were created with if they were created with a block size that is not an increment of 28 words.
- For disk mass storage files, block size should be a multiple of the prep factor (28, 56, or 112) to eliminate the Executive read-before-write.
- A number of systems processors (such as ED and DATA) and symbionts require a block size of 224 words.
- Mass storage files that were created with a block size that is an increment of 28 words may be read with a different block size. However, this block size must be an increment of 28 words.
- I/O overhead can be reduced by choosing a block size which is greater than or equal to the record size (or a multiple of it), and which is a multiple of the prep factor.

G.2.1.4. SDF End-of-File Record

The SDF end-of-file record is written in the last word of the last block. It has the form:

T1	T2	S5	S6
07700	<i>p</i>	0	0

where *p* is T1 from the previous image control word.

G.2.1.5. SDF File Layout

The SDF file layout may have one of two forms:

Sequential SDF File on Mass Storage*		Direct SDF File on Mass Storage	
Sector 0	label record	Sector 0	label record
	data record #1		bypass record
	data record #2		bypass record
	data record #3		bypass record
8	data record #4	4	record #1
	data record #5		record #2
	data record #6		record #3
	data record #7		record #4
	bypass record		record #5
	bypass record		record #6
16	EOF record		record #7
			EOF record

* For SDF sequential tape files, the file may be blocked as shown with eight sectors per block.

G.2.2. SDF File Processing

When the FORTRAN compiler encounters an I/O statement, it generates an I/O packet and a call to the FORTRAN I/O processor interface module (in C2F\$ or in a relocatable library). The interface module does a lookup in the File Reference Table (FRT) using the unit reference number as an index. The FRT entry has a pointer to the FCT if that unit has already been opened. This pointer is zero if the file is not open. If the pointer is zero, the interface module gets space from the free core area for the FCT and the buffers. The interface module generates the FCT using the information given on the OPEN or DEFINE FILE statement, the input label, or default information. The interface module calls on PCIOS to open the file for input or output when required. The processor interface module also calls on PCIOS to actually read, write, backspace and close (endfile) when specified. Record editing is done in the ASCII FORTRAN library.

G.2.2.1. Sequential Access

Records are input/output using a double buffering process. Records are generated in an intermediate record area on output and then buffered to the file device. On input, records are processed within the input buffer if they are contained within the buffer or moved to the intermediate record area if they are not. When records are buffered out to the file they are automatically segmented at the segment size specified in the FCT.

The intermediate record area must be large enough to handle the largest formatted, list-directed or namelist record it may encounter or the largest segment it may encounter for unformatted records. Initially this area is set to the default maximum record size or default segment size specified in the SCT (see G.8), whichever is larger. When an OPEN or DEFINE FILE statement is encountered with a larger maximum record size specification or a larger segment size specification, this area is freed and another area is obtained from the free main storage area.

Unformatted records are processed by segments. Therefore, the maximum record size field of the OPEN or DEFINE FILE statement does not apply. The segment size specified in the FCT determines the size of the segments.

The programmer must specify the maximum record field on the OPEN or DEFINE FILE statement to be able to write formatted records larger than the default record size specified in the SCT.

The ENDFILE statement will write the SDF end-of-file record in the file and write the last block on the file device. Each block in an SDF file is the uniform block size specified for that particular file. This uniformity is accomplished by filling the last block with bypass records and writing the end-of-file record as the last word in the last block. After an ENDFILE statement, the file is positioned so that a WRITE statement will begin another SDF file on the device. WRITE, REWIND, BACKSPACE, and ENDFILE (creates an empty file) are allowed after an ENDFILE.

The BACKSPACE statement backspaces over all segments of a record if the record is segmented.

G.2.2.2. Direct Access

Direct access processing uses the record number to compute the location of the record within the file. The record is then read into a buffer unless the record already resides in the buffer due to a previous read.

The buffer size specified by the BUFR clause of the OPEN statement is used to determine the number of records read/written on an EXEC I/O request. A larger buffer size will reduce the number of I/O requests needed in cases where record access tends to be localized.

If skeletonization is specified when a direct access file is initially created, dummy records will be written to the file resulting in all necessary storage being acquired if not already acquired. The dummy records, if not written over by subsequent WRITE statements, will be recognized as nonexistent records on subsequent read statements. Detection of a dummy record on a direct access read is indicated by error code 1053.

If skeletonization is not specified when a direct access file is initially created, no dummy records are written to the file. Nonexistent records are not recognized on a direct access read of a nonskeletonized file. In fact, if only a maximum file size is provided when assigning the file, storage for areas of the file not yet written to will not actually be acquired until a write to that area takes place.

If nonexistent records must be recognized on a read of a direct access file, the file must be skeletonized when initially created and when extended.

Reading areas of a nonskeletonized direct access file before they have been written to may result in an I/O error with a status of 5 if the area meant to hold the requested record has not yet been acquired.

Direct access records are only segmented when the records are longer than 2,047 words.

Direct access SDF files may be read using sequential I/O statements. Records will be read in ascending record number order. Only records previously written (that is, bypass and dummy records are ignored) will be returned.

G.2.3. SDF Files Not Written by Processor Common I/O

SDF files not produced by the Processor Common I/O System such as FORTRAN V SDF files may be read sequentially with ASCII FORTRAN if the data within the record is compatible with ASCII FORTRAN. These files may only be read. Backspacing is not allowed.

No attempt is made to determine what type of record is being used. If an unformatted read is specified, the record is treated as an unformatted record. If a formatted read is specified, the record is treated as a formatted record.

Formatted records are read into the intermediate record area and if the SDF label specifies Fielddata (bit 0 of the label control word = 0), the complete record is translated to ASCII before the record is edited. If the file has formatted records larger than the default maximum record size, an OPEN or DEFINE FILE statement must be used.

If the file has segmented records, it is assumed that they are segmented using the SDF continuation control image (051). For formatted reads, all segments of the record are read into the record area before the record is edited. For unformatted reads, the record will be processed by segments. If unformatted records are to be read which are larger than the default segment size specified in the SCT, the OPEN or DEFINE FILE statement must be used to indicate this.

The files may reside on tape or mass storage. If mass storage contains stacked files (WRITE, ENDFILE, WRITE), only the first file can be read. These files can be copied to tape with the B option and then processed as tape files.

G.3. ANSI Magnetic Tape Interchange Format

The ANSI file formats produced by ASCII FORTRAN comply with the American National Standards Institute (ANSI) magnetic tape interchange format as described in the American National Standards proposal X3L5/365T, dated September 27, 1973, and titled "Magnetic Tape Labels and File Structure for Information Interchange".

G.3.1. ANSI File Description

An ANSI file may be labeled or unlabeled. If the J option is present on the assign of the tape, an unlabeled ANSI file will be read or written. If the J option is not present, a labeled ANSI file will be read or written.

A labeled ANSI file produced by ASCII FORTRAN contains a header label group, data blocks and a trailer label group. Data blocks are separated from the header and trailer labels by a tape mark. Files on a tape are separated by a tape mark.

The header label group consists of the VOL1 Label, HDR1 label and the HDR2 label. The trailer label group consists of the EOF1 label and the EOF2 label. If a file spans tape reels, the EOVI and EOVI labels terminate all file sections except the last file which is terminated by the EOF1 and EOF2 labels.

Labels are written using the Executive tape labeling facilities (TLS). The volume labels are only present in the first header label group of each tape and are automatically processed by TLS.

Labeled ANSI magnetic tapes have the following structures. Asterisks (*) indicate tape marks.

- For a single file, single volume:

VOL1 HDR1 HDR2* [Data Block 1] [Data Block 2] [Data Block 3] * EOF1 EOF2**

- For a single file, multivolume:

VOL1 HDR1 HDR2* [Data Block 1] [Data Block 2] [Data Block 3] * EOVI EOVI**

VOL1 HDR1 HDR2* [Data Block 4] * EOF1 EOF2**

- For a multifile, single volume:

VOL1 HDR1 HDR2* [Data Block 11] [Data Block 12] * EOF1 EOF2**

HDR1 HDR2* [Data Block 21] [Data Block 22] * EOF1 EOF2**

- For a multifile, multivolume:

VOL1 HDR1 HDR2* [Data Block 11] [Data Block 12]

[Data Block 13] * EOF1 EOF2 * HDR1 HDR2* [Data Block 21] * EOVI EOVI**

VOL1 HDR1 HDR2* [Data Block 22] [Data Block 23] * EOVI EOVI**

VOL1 HDR1 HDR2* [Data Block 24] * EOF1 EOF2* HDR1 HDR2* [Data Block 31] * EOF1 EOF2**

An unlabeled ANSI file produced by ASCII FORTRAN allows the use of the ANSI record formats by sites which do not have TLS.

All of the ANSI record formats are available on output when creating an unlabeled ANSI tape with ASCII FORTRAN. Creation of a null or empty unlabeled ANSI tape file will not be permitted.

On input, all ANSI record formats may be read provided the unlabeled ANSI tape was created by ASCII FORTRAN or the data on the unlabeled ANSI tape meets the requirements of the ANSI standards. Reading data blocks from unlabeled foreign tapes is only possible when using the U, F, or FB record formats.

Since there are no labels, PCIOS is unable to compare record size, block size, buffer offset, and record format. It is the user's responsibility to ensure that the values passed to PCIOS (via the OPEN and DEFINE FILE statements) are correct for the file to be read.

The structure of an unlabeled ANSI magnetic tape is:

- For a single file, single volume:

`Data Block 1` `Data Block 2` `Data Block 3` **

- For a single file, multivolume:

`Data Block 1` `Data Block 2` `Data Block 3` * `Swap Block` ** `Data Block 4` **

- For a multifile, single volume:

`Data Block 11` `Data Block 12` * `Data Block 21` `Data Block 22` **

- For a multifile, multivolume:

`Data Block 11` `Data Block 12` `Data Block 13` * `Data Block 21` * `Swap Block` **

`Data Block 22` `Data Block 23` * `Swap Block` ** `Data Block 24` * `Data Block 31` **

The user may specify the following record formats using the OPEN statement described in 5.10.1 or the sequential DEFINE FILE statement described in 5.6.6. The data block format is dependent on the record format and blocking factor chosen.

<u>Format</u>	<u>Description</u>
Undefined	The record is written with no control information appended to it. The records may be variable in length. Each data block contains only one record.
Fixed-unblocked	The record is written with no control information appended to it. All records are of the same length. Each data block contains only one record.
Fixed-blocked	Basically the same as fixed-unblocked; however, each block contains one or more records. Each data block has the same number of records except possibly the last block. The last block is truncated if it does not contain the full number of records.
Variable-unblocked	The record is written with four characters of control information indicating the record length in ASCII characters appended to it. The records may be variable in length. The data block contains only one record.

Variable-blocked	The same as variable-unblocked except that the block will contain as many complete records as possible.
Variable-unblocked-segmented	The record is segmented at the block boundary. Appended to each record segment are five characters of control information. The first character is the record segment indicator and the remaining four are the segment length. The data block contains only one segment of the record.
Variable-blocked-segmented	Similar to variable-unblocked-segmented; however, each block may contain more than one record segment but never more than one segment of the same record.

The segment indicator character has the following meaning:

<u>Value</u>	<u>Meaning</u>
0	Record begins and ends in this segment
1	Record begins but does not end in this segment
2	Record neither begins nor ends in this segment
3	Record ends but does not begin in this segment.

ASCII FORTRAN I/O does not produce any control information on the front of the data block; hence the buffer offset field of the HDR2 label is set to zero. Files with such information can be read with ASCII FORTRAN but the OFF clause of the OPEN statement or the buffer offset field of the DEFINE FILE statement must indicate the size in characters of this control information. No attempt is made to interpret this control information. It is skipped over when reading the file.

ANSI files are treated as character data files and are processed internally in quarter-word mode. The actual format of the data written on the tape depends upon whether or not the tape is being written through an MSA or not.

To produce an ANSI file that is to be interchanged, an MSA must be available and the quarter-word (MSA A) format must be specified in the format field of the ASG control command. The programmer may specify the 6-bit packed (MSA B) format or the 8-bit packed (MSA C) format but the tapes will not be interchangeable because of the way the data is written to the tape. The programmer specifies which format he desires in the format field of the ASG control command. The format specified when the tape is written must also be specified when the tape is read. If an MSA is not available, the default formats are 8-bit packed for 9-track tapes and 6-bit packed for 7-track tape.

Block padding may be done on output depending upon the format with which the tape is being written. The pad character is the ASCII circumflex.

<u>Format</u>	<u>Result</u>
quarter-word	No padding is done since the block is truncated at the last data character of the data block.
8-bit packed	The block may contain 0, 1, 2, or 3 pad characters since an increment of words must be written.
6-bit packed	Blocks may contain 0, 1, 2, or 3 pad characters since an increment of words must be written.

G.3.2. ANSI File Processing

ANSI files are processed much the same as SDF files with the following exceptions:

- ANSI records are always moved to the intermediate record area on input and generated in the intermediate record area on output. This means that unformatted records are not processed by segments so the maximum record field of the OPEN or DEFINE FILE statement must reflect the largest record being read or written whether it is formatted or unformatted.
- ANSI records are moved to and from the buffers character by character rather than by words.
- The BACKSPACE request must not be specified if the records are blocked.
- Unformatted records must not be written to a file using quarter-word (MSA A) format unless all the variables are character type, since this format strips the ninth bit of each quarter word. Therefore, if this bit is not zero, the block is truncated.

G.3.3. ANSI Interchange Tapes from Other Systems

ANSI Interchange tapes written on other systems must be read through an MSA in quarter-word mode. If read with the other formats (8-bit or 6-bit) the data would not be aligned in the read buffer correctly.

These ANSI files may have control information in the front of the data block. If so, this must be specified in the OFF clause of the OPEN statement or the buffer offset field of the DEFINE FILE statement. This control information is ignored and can be from 1 to 99 characters long.

These files may contain pad characters (ASCII circumflex) after the data in the data block, but the last character of a data record must not be an ASCII circumflex pad character.

Labels other than HDR1, HDR2, EOF1, EOF2, EOVI, or EOVI2 are ignored.

G.4. ASCII Symbiont Files

In order to read symbiont input files or write symbiont output files, the user must dedicate certain unit reference numbers to these files. This is done by generating the file reference table (FRT) (see G.6) or by using an OPEN or DEFINE FILE statement. The unit reference number dedicated to the particular type of symbiont file desired is referenced in input/output statements.

OPEN, READ, INQUIRE, and CLOSE are the only I/O statements allowed for an input symbiont file.

OPEN, WRITE, ENDFILE, INQUIRE, and CLOSE are the only I/O statements allowed for output symbiont files. ENDFILE is allowed only for the APUNCH\$ and APNCHA\$ files and causes an @EOF card to be punched.

Queuing of alternate print (APRNTA\$) and punch (APNCHA\$) files for symbiont processing is done as follows:

- If the alternate file is assigned to the run as a new file prior to execution of the FORTRAN program with an @ASG,C statement, the FORTRAN I/O will do a @BRKPT when the file is closed. It is up to the user to do the @FREE and @SYM to have it printed or punched.
- If the alternate file is assigned to the run with the @ASG,A statement prior to the execution of the FORTRAN program, the FORTRAN I/O will do a @BRKPT when the file is closed. It is up to the user to do the @SYM to have it printed or punched.
- If the alternate file is not assigned to the run when a batch FORTRAN program is executed, the FORTRAN I/O will not do an assignment but will allow the Executive to assign the file when the first ER APRNTA\$ or ER APNCHA\$ is done. (See the EXEC Programmer Reference, UP-4144.3 (see Preface).) The FORTRAN I/O will direct output to an alternate print file via @BRKPT when the file is closed. The Executive will automatically queue the file for printing or punching when the breakpoint is done. Note that in demand the user must assign the alternate file.

The default symbiont record sizes are as follows:

- APRINT-APRNTA-AREADA - 132 characters
- APUNCH-APNCHA - 80 characters
- AREAD - 80 characters

These defaults may be changed by the OPEN or DEFINE FILE statement.

G.5. Unit Reference Number and File Assignment

The FORTRAN language refers to files through a unit reference number. The Series 1100 Operating System maintains files under an external file name of the form:

[[*qualifier* *] *file-name* [(*F-cycle*)] [/ [*read-key*]] [/ [*write-key*]]

See the current EXEC Programmer Reference, UP-4144.3 (see Preface) for a description of the external file name.

The unit reference number must be linked to the external file name before the file can be referenced. Note that the external file may not be word-addressable mass storage (equipment types 020-027) because of a PCIOS restriction. The unit reference number may be linked to the external file name in one of three ways:

- If the FILE clause is present in the OPEN statement, a @USE *unit reference number*, *file-name* will be performed at execution time. The file name in the FILE clause is restricted to the form:

[*qualifier* *] *file-name* [.]

- Use the unit reference number as the *file-name* when the file is assigned:

@ASG,T 4,U,BLANK

- Use the unit reference number as an alternate or internal file name by linking it with the external file name using the USE command:

@ASG,A FILEX
@USE 4,FILEX

When a FORTRAN program that refers to a file is executed, that file may already be assigned to the run, cataloged, or nonexistent.

If the file is already assigned, the unit reference number must be linked as described previously.

If the file is cataloged and not assigned to the run, it will automatically be assigned by the FORTRAN I/O Module if the file is not rolled out. The file must be cataloged with the unit reference number as the *file-name* or, if the FILE clause of the OPEN statement is not used, a USE command must have been performed prior to execution linking the unit reference number with the cataloged file name.

If the file is neither assigned nor cataloged, a temporary mass storage file of 128 tracks will be assigned for sequential files by the FORTRAN I/O Module. For direct access files, the track size is determined from the OPEN statement clauses or DEFINE FILE statement parameters, and may be less than 128 tracks. If the location in the file reference table designates this as a symbiont APRINT\$, APUNCH\$, or AREAD\$ file, then no file assign is attempted since file space for these symbionts is handled by the Executive.

Once a file reference number is associated with a file and that file is opened, the file reference number may not be associated with another file until the first file is closed via either the CLOSE statement or the CLOSE service subroutine.

G.6. File Reference Table Element – F2FRT

The element F2FRT is the file reference table (FRT). The FRT is a variable length table of one-word entries which is used to link the unit reference number with the file control table for the physical file being processed. The unit reference number is used as an index into the FRT.

The entries in the FRT contain a pointer to the file control table if the file has been opened or zero if the file is not open. In the latter case, an automatic open will occur when that particular unit is referenced with an I/O statement. The FRT entry may also contain a code in S2 designating the unit as a symbiont file as follows:

<u>Code</u>	<u>File Indicated</u>
051	APRINT\$ symbiont
052	APUNCH\$ symbiont
053	AREAD\$ symbiont
054	APRNTA\$ alternate symbiont
055	APNCHA\$ alternate symbiont
056	AREADA\$ alternate symbiont

The format of the file reference table is:

	S1	S2	S3	H2
-3	0	053	a	f
-2	0	052	a	f
-1	0	051	a	f
F2FRT\$+0	0	s	a	f
1	0	s	a	f
2	0	s	a	f
3	0	s	a	f
4	0	s	a	f
n	0	s	a	f

where:

f is set to zero until a reference is made to the unit. Then it contains the address of the file control table.

a indicates the file status.

<u>Value</u>	<u>Status</u>
0	Never referenced
1	Open
2	Closed
3	Reread unit

s is a symbiont code or zero.

F2FRT -1, -2, -3 are used for those I/O statements that do not have a unit designated, implying one of the symbionts APRINT\$, APUNCH\$, or AREAD\$.

An installation may generate a new file reference table by remaking the F2FRT element as follows:

```
@PDP FTNPROC,FTNPROC
@MASM,SI F2FRT,F2FRT
  F$FRT      f      . Procedure call
  PR         |      .
  PU         |      .
  CR         |      .
  APR        |      .
  APU        |      .
  ACR        |      .
  RR         |      .
  END
```

where:

- f* is the largest unit reference number to be accessed; PR, PU, CR, APR, APU and ACR specify the symbionts APRINT\$, APUNCH\$, AREAD\$, APRNTA\$, APNCHA\$, and AREADA\$, respectively. RR specifies that the unit is to be a reread unit.
- /* is a unit reference number list of the form u_1, u_2, \dots, u_n . Each u must be in the following range: $0 \leq u \leq f$. The units specified will be designated as symbiont files of the type specified in the operation field (PR, PU, etc.) or as a REREAD unit. If a unit is specified in more than one operation it will get the designation of the last operation field.

The operation fields PR, PU, etc. may be given in any order, but F\$FRT must always be the first operation.

When an entry in the file reference table is designated as one of the symbiont types above, it must be used for that type of symbiont file unless an OPEN statement is used to specify another file type.

F2FRT is released with the following designation:

```
@PDP FTNPROC, FTNPROC
@MASM,SI F2FRT,F2FRT
  F$FRT      30
  PR         6
  PU         1
  CR         5
  RR         0
  END
```

The OPEN statement may be used to override and modify the symbiont code field specified when the file reference table was built provided the unit had never been opened before or had a file status of closed before the OPEN statement. The new code (or zero for SDF and ANSI) will remain in effect for the duration of the run or until reset by another OPEN statement.

G.7. Free Core Area Element - F2FCA

Storage for record areas, buffers, file control tables and floating-point conversion is obtained from the free core area when required. A scratch area must be added when using the Sort/Merge interface (see L.3.3 and the CORE= clause in L.4.1). The free core area is generated as follows:

```
@PDP FTNPROC,FTNPROC
@MASM,SI F2FCA,F2FCA
  F$FCA      s      . procedure call
END
```

where s is a number indicating the amount of storage required. It may be 0 or void. The amount of space required can be calculated from the following formula:

$$s = a + b_1 + b_2 \dots + b_n + c$$

VariablesDetermination

$$a = 1 + 2 * m_1 + (m_2 + m_3) * (\text{default-segment-size})$$

where:

$m_1 = \text{default-segment-size}$ or largest record size, whichever is larger.

$m_2 = 2$, if any DEBUG statements are present in the FORTRAN program; otherwise, $m_2 = 0$.

$m_3 = 2$, if any calls to PDUMP or FTNPMD or any F-option FTN compilations are present in the FORTRAN program; otherwise, $m_3 = 0$.

b There will be a b for every file active and the size of b depends upon the type of file (SDF, ANSI or symbiont).

For symbiont files:

$$b = 64 + 1$$

For SDF direct files:

$$b = 64 + 2 * (\text{record-size} + 1) + 2 * \text{prep-factor}$$

For SDF sequential files:

$$b = 64 + (2 * (\text{block-size-in-words}) + 1) + 1$$

For ANSI files:

$$b = 64 + ((2 * (\text{block-size-in-chars} + 3)) / 4) + 1$$

c Miscellaneous core requirements:

$$c = 25 \text{ if no scratch area is needed for the Sort/Merge interface}$$

Otherwise:

$$c = 25 + \text{the amount of scratch area needed for the Sort/Merge interface}$$

If s is void or zero, no space will be reserved. Instead, ER M CORE \$ will be performed to add space to the end of the control D-bank as required.

The default record size, block size, and segment size are given by parameter to the F2SCT element (see G.8).

The standard F2FCA, as distributed by Sperry Univac, is released with no space reserved, so the M CORE \$ method is used.

When s is coded other than zero, or is void, the space reserved must be sufficient to fill a core request or the program will terminate in error.

If any subroutines which are not written in ASCII FORTRAN are called by an ASCII FORTRAN program, these subroutines should not allocate and deallocate storage. If any M CORE \$ or L CORE \$ is to be executed by any of these subroutines, element F2FCA should be generated with the size of storage needed by the ASCII FORTRAN program.

G.8. Storage Control Table Element – F2SCT

The element F2SCT is the storage control table which contains register save areas, working storage and pointers to the file reference table and the free core area header. In addition, F2SCT contains a location that contains the default record size, the default segment size, and the default block size for SDF sequential files. These default fields may be changed at an installation by remaking the F2SCT element as follows:

```
@PDP FTNPROC,FTNPROC
@MASM,SI F2SCT,F2SCT
  F$SCT  $x,y,z$  . procedure call
  END
```

where:

- x is the default SDF record size desired in words. The range is $1 \leq x \leq 262,000$ but it must be remembered that a core area of size $2 * x$ is reserved for the processing of records.
- y is the default SDF sequential segment size desired in words. The range is $1 \leq y \leq 2,047$.
- z is the default SDF sequential block size desired in words.

F2SCT is shipped with the following default sizes:

```
 $x = 33$ 
 $y = 111$ 
 $z = 224$ 
```

G.9. Input/Output Errors

When an input/output error is detected by the ASCII FORTRAN I/O handler, the action taken depends upon whether or not an ERR clause (see 5.2.6) contingency was specified and whether or not an I/O status clause specification (see 5.2.8) was present.

G.9.1. ERR Clause Specified

If an ERR clause is specified and an error or warning condition is encountered, the I/O status word PTIOE in the storage control table is set and transfer is made to the statement specified by the ERR clause. In addition, if the I/O status clause is present, the IOSTAT variable receives the contents of PTIOE prior to the transfer. An error message is not printed.

The I/O status word is set as follows:

Q1	Q2	H2
<i>s</i>	<i>u</i>	<i>c</i>

where:

c indicates the cause of the error. Its value is the integer clause number specified on the error clauses listed in G.9.3.

s indicates the substatus of the error if any.

If *c* is 1 then *s* is the I/O status coded from the I/O packet.

For ANSI files, if the error involves an ER to TLBL\$, the ER TLBL\$ error code will be returned in the substatus field.

For those errors that have no substatus, *s* is 0.

u is the integer file reference number of the file in error:

<u><i>u</i></u>	<u>File</u>
$0 \leq u \leq f$	The unit number, where <i>f</i> is the largest file reference number allowed (see G.5)
-1	The APRINT\$ symbiont
-2	The APUNCH\$ symbiont
-3	The AREAD\$ symbiont
-0	The unit is undetermined

The particular fields of the I/O status word may be tested using the functions:

IOC()	cause
IOS()	substatus
IOU()	unit

In addition, if the I/O status clause was present, the particular fields are available via the IOSTAT variable.

G.9.2. ERR Clause Not Specified

If an ERR clause is not specified, but an I/O status clause is specified:

- the I/O status word PTIOE is set;
- the IOSTAT variable receives the contents of PTIOE; and
- control is immediately transferred back to the program if an error is detected. If a warning condition is detected, execution of the current statement is continued.

If an ERR clause and an I/O status clause are both not specified:

- an error or warning message is printed; and
- for errors, all opened files are closed, and the program is terminated;
- for warnings, execution of the current statement is continued.

The error message has the form:

```
FTN ERR ON UNIT u c
```

where:

- u* may be a positive unit number, PRINT, PUNCH, READ, or ILLEGAL.
- c* is one of the error clauses listed in G.9.3. For some error clauses, a second line is printed to provide additional information regarding the error.

G.9.3. Error Clause Listing

When the Common Storage Management System detects error conditions, it prints one of the following error messages:

```
STORAGE FULL  
BAD FREE ADR  
BAD FREE LEN  
BAD CHAIN
```

Note that these error messages have no effect on PTIOE or the IOSTAT variable.

The following two lists are of the warning and error clauses applicable to the ASCII FORTRAN I/O error messages. The number listed is the number set in the clause field of the I/O status word PTIOE. The first list gives the errors detected by PCIOS; the second list gives the errors and warnings detected by FORTRAN library routines.

<u>Number</u>	<u>Message</u>
1	<p>I/O STATUS CODE <i>xx</i></p> <p>An ER in PCIOS received the nonzero I/O status code upon completing an I/O request (see the EXEC Programmer Reference, UP-4 144.2 (see Preface)).</p>
6	<p>MAX REC SIZE NOT CONSISTENT WITH FILE</p> <p>The C2DSDF module returned a bad status while trying to open a file for direct input.</p>
9	<p>INVALID FILE STRUCTURE</p> <p>The C2SSDF module returned a bad status while trying to backspace a file opened for sequential access. The file originator was not PCIOS.</p>
10	<p>INVALID DATA BLOCK STRUCTURE</p> <p>The C2SSDF module returned a bad status while trying to backspace a file opened for sequential access.</p>
12	<p>FILE NOT SDF DIRECT</p> <p>The C2DSDF module returned a bad status while trying to open a file for direct input.</p>
13	<p>FILE NOT CONSISTENT WITH DEFINE FILE</p> <p>The C2ANSI module returned the error while trying to open the file for input.</p>
16	<p>FILE NOT SDF</p> <p>The C2SSDF module returned the error while trying to open the file.</p>
37	<p>BLOCKSIZE NOT MULTIPLE OF FIXED RECSIZE</p> <p>The C2ANSI module returned the error while trying to read (write) from (to) the file.</p>
39	<p>INCORRECT DATA BLOCK COUNT</p> <p>The C2ANSI module returned this error while executing a tape swap or a close on the file.</p>
50	<p>ERROR ON TBL\$ REQUEST</p> <p>The C2ANSI module returned this error while attempting an open, close, or tape swap on this file</p>
65	<p>MASS STORAGE FILE OVERFLOWED</p> <p>The C2SSDF module returned this error while attempting a write to the file.</p>

66 BLOCKSIZE INCONSISTENT WITH FILE

The C2SSDF module returned this error while attempting to open the file for sequential access.

67 INCORRECT VARIABLE FORMAT CONTENTS

68 FILE LABEL LACKS EXTENDED PARAMETERS

69 FILE STRUCTURE LACKS AN END-OF-FILE

70 MAXIMUM INTERCHANGE RECORD SIZE EXCEEDED

71 TAPE LABELING SYSTEM NOT AVAILABLE

72 LOSS OF POSITION ON THE TAPE UNIT

73 SKELETONIZATION INCONSISTENT WITH FILE

74 EXTENDED FILE SMALLER THAN ORIGINAL FILE

C2DSDF returned the error when the size of the file to be extended is specified to be smaller than the size of the original file.

The following errors and warnings are detected by FORTRAN library routines. Asterisks (*) indicate warnings. The double asterisk (**) preceding clause 1015 indicates the mistake is fatal for nonformatted records and list-directed DECODE records, and is a warning for other formatted records.

NumberMessage

1001 INVALID RECORD NUMBER

The record number specified in a direct access READ, WRITE, or FIND statement is not in the range $1 \leq \text{record number} \leq \text{max/rcd/num}$, where max/rcd/num was specified in a direct access OPEN or DEFINE FILE statement.

1002 INAPPROPRIATE UNIT

The unit number specified in the I/O statement is bad for one of the following reasons:

- The unit number specified was not in the range $0 \leq \text{unit number} \leq \text{largest unit number to be referenced}$. Refer to G.6 for a discussion on the largest unit number to be referenced.
- An attempt was made to read from a printer, write or punch to a card reader, etc.
- The file reference table has been overwritten.

1003 FILE ASSIGNMENT FAILED

One of the following file assignments failed:

- ASG,A of a file whose status was specified as OLD in an OPEN statement.
- ASG,T of a file whose status was specified as NEW, SCRATCH or UNKNOWN in an OPEN statement or the file was being opened by a statement other than the OPEN statement.
- A temporary file could not be obtained in an HVTS environment.
- The requested cataloged file is rolled out.

1004 ATTEMPTED OPEN RANDOM ON TAPE FILE

An attempt was made to open a tape file as a direct access file.

*1005 ILLEGAL FORMAT CHARS - TREATED AS BLANKS

The warning message is given only once per statement. It is given for one of the following reasons:

- The format given to the run-time library routines in an array or character expression contains an unknown format type or an illegal combination of values. The illegal combination could be *Aw.d* since *.d* cannot be used with an *Aw* format. The unknown format type would be any alphabetic character not used as a format type now or it could mean a missing alphabetic character when one should occur, such as *w.d*. The scanning of the format array or expression will continue following the warning unless too many errors have been found.
- The values in a FORMAT statement may have been encoded badly by the compiler. The encoded FORMAT contains an illegal code. The scanning of the FORMAT will continue after the warning is given.

1006 UNIT NOT OPEN FOR RANDOM INPUT/OUTPUT

A direct access READ, WRITE, or FIND statement specified a unit (other than one attached to a symbiont) that had been opened for sequential access.

1007 UNIT NOT AVAILABLE FOR BUFFERED I/O

One of the following statements attempted to access a symbiont:

- Unformatted READ or WRITE
- REWIND
- BACKSPACE
- Direct access READ, WRITE, or FIND

1008 FILE ASG'D CAN'T HOLD ALL DIRECT RECORDS

A user-assigned file is not large enough to hold all of the records requested in the direct access OPEN or DEFINE FILE statement.

1009 UNIT NOT AVAILABLE FOR SEQUENTIAL I/O

One of the following statements attempted to manipulate a file opened for direct access.

- A sequential DEFINE FILE, READ, or WRITE
- ENDFILE
- REWIND
- BACKSPACE

1010 READ TYPE AND RECORD TYPE DO NOT CONFORM

The type (formatted, unformatted, list-directed, namelist) of a sequential or direct access read differs from that of the record to be read.

1011 ATTEMPTED READ AFTER WRITE

An attempt was made to read a record from a sequential file which is currently open for output.

1012 ENDFILE ONLY LEGAL FOR PUNCH SYMBIONTS

The only symbionts for which an ENDFILE can be executed are PUNCH\$ and APUNCH\$.

1013 READ/WRITE TYPE INHIBITED FOR THIS FILE

An attempt was made to execute a formatted (unformatted) read or write on a direct access file which contains only unformatted (formatted) records.

1014 ATTEMPTED TO READ PAST AN END-OF-FILE

**1015 RECORDS EXCEEDING MAX LENGTH ARE FAULTY

During a formatted read or write, the format cannot read or write more than the record length. This warning message is given if the format requires a length greater than the given record length.

During an unformatted read or write of direct access files, the read or write attempts to require more length than the record contains. The message is given for this fatal error.

During an unformatted write of ANSI files, the write attempts to require more length than the record contains. The message is given for this fatal error.

During list-directed DECODE statements, the size of the DECODE block or internal storage area is exceeded.

***1016** **FORMAT TYPE NOT SAME AS INTERNAL TYPE**

The type of the input/output list item is the internal type. The format type given would require a conversion to store the value to the list item on input or to write the list item value on output. A conversion is not done if the types do not match, that is, *Ew.d* format with an integer list item. The run-time routine will give a warning message and assume that it can try and use the given value with that particular format code.

1017 **ABSOLUTE VALUE OF I/O ITEM TOO LARGE**

The absolute value of the exponent of a floating-point number is too large. The number of input digits for an integer exceeds the amount that may be used to hold the number internally. The fatal error message is given to prevent an overflow during conversion of the number during input or output routines.

***1018** **INPUT DATA DOES NOT CORRESPOND TO TYPE**

The warning message is given when the data being read is not consistent with the type of the input list item. Up to 132 characters of the record in error will be printed.

Floating-point any characters other than numerals, plus, minus, and blank, or an incorrect form of a value.

Octal any characters other than numerals or blanks.

Integer any characters other than numerals or blanks.

1019 **NAMelist NAME HAS MORE THAN SIX CHARS**

- The namelist name in the input data is longer than the legal six characters.
- The variable name in the input data in the present namelist is longer than six characters.

1020 **INCORRECT CHARACTER IN NAMelist INPUT**

The error is given when:

- the first nonblank character following the namelist name in the input is not an alphabetic character for a variable name;
- the subscript of the variable name is not followed by an equal sign;
- the length for a Hollerith input field is negative;
- the input data is Hollerith but the namelist variable is not real or integer;
- the input data is literal but the namelist variable is not of type character;
- the variable name is missing before the equal sign in the input data;
- the input data is octal but the list item is logical; or
- the input subscript contains something other than numerals, plus, minus (if lower bounds are present), left and right parentheses, commas, and blanks.

1021 NAMELIST INPUT HAS TOO MANY SUBSCRIPTS

The subscript of the input variable name contains more than seven dimensions.

*1022 NAMELISTS DO NOT CONTAIN VARIABLE NAME

The variable name in the input data is not part of the present namelist. This is a warning message. The namelist read will skip this name and continue reading at the next variable name.

1023 UNIT NOT CONSISTENT WITH FILE FORMAT

The inconsistent file format arises in one of the following situations:

- A sequential ANSI OPEN or DEFINE FILE is attempted and the associated file is not a tape file.
- A sequential DEFINE FILE is attempted and either the DEFINE FILE statement or the Define File Block, if present, attempts to override a predefined (within the FRT) symbiont file format.

1024 LIST NOT SATISFIED BY LAST FORMAT GROUP

The FORMAT is not an empty format but does not contain any repeatable editing codes to handle the items in the input/output list. This is a fatal error message.

1025 UNABLE TO FILL SPACE REQ FROM FREE CORE

Insufficient free storage area exists to satisfy the dynamic storage requirements of a run-time routine. Refer to G.7.

1026 INCORRECT VARIABLE FORMAT CONTENTS

During ENCODE/DECODE before ASCII FORTRAN level 8R1, the message was produced for list-directed ENCODE/DECODE statements, which were not allowed.

If a format in an array or character expression gets more than ten errors, the fatal diagnostic is given. If the size of w , d , e , or p is too large for the field width, this fatal diagnostic occurs.

1027 FILE ALREADY OPEN WITH ANOTHER UNIT

An attempt has been made to open a file which is already associated with an opened unit different from the one specified in the OPEN statement. A file may not be associated with two opened units at the same time. A second line to the error message will indicate the file involved.

1028 INVALID LITERAL IN OPEN/CLOSE CLAUSE

A literal argument appearing in one of the clauses of an OPEN or CLOSE statement is either misspelled or invalid.

1029 NEGATIVE RECORD LENGTH SPECIFIED ON OPEN

The record length specified in the RECL or MRECL clauses of the OPEN statement is negative.

1030 OPEN/CLOSE/INQ OPTION MISSING OR ILLEGAL

One of the errors listed below was encountered in an OPEN, CLOSE, or INQUIRE statement. A second line to the error message will indicate which error condition occurred.

- In an OPEN statement, if a file is being opened for unformatted I/O, the BLANK clause may not be present. For this case, the second line to the error will be:

CLAUSE(S) IN ERROR — BLANK

- The RECL clause is missing on an open of a direct access file. For this case, the second line to the error message will be:

CLAUSE(S) IN ERROR — RECL

- Clauses which pertain only to a direct (sequential) open were present on a sequential (direct) open. See 5.10.1. For this case, the second line to the error message will be:

CLAUSE(S) IN ERROR — DIRECT

If sequential clauses were present on a direct open, the following message will appear:

CLAUSE(S) IN ERROR — SEQUENTIAL

- The FORM and RFORM were both present in an OPEN statement. For this case, the second line to the error message will be:

CLAUSE(S) IN ERROR — FORM & RFORM

- An open status of SCRATCH may not be present if the file clause is present. For this case, the second line to the error message will be:

CLAUSE(S) IN ERROR — STATUS

- A zero length literal was used as an argument in an OPEN, CLOSE, or INQUIRE statement.

1031 OPEN STATUS OLD, FILE DOES NOT EXIST

An open status of OLD or EXTEND was specified in an OPEN statement; however, the file could not be assigned to the run.

1032 ASG,CP ASSIGNMENT FAILED ON OPEN

An open status of NEW was specified in an OPEN statement; however, the file was not preassigned by the user and one could not be assigned.

1033 FILE INCONSISTENT WITH DEFINE FILE/OPEN

The file format specified in an OPEN or DEFINE FILE of a file already opened did not match the file format of the opened file.

1034 SPECIFIED INPUT BEYOND END OF RECORD

The unformatted read requires more length from the record than was actually present. The record may or may not have been segmented. All segments would have been read if the record was segmented.

1035 SCRATCH FILE NAME CONFLICT

An open status of SCRATCH was specified in an OPEN statement. In attempting to assign a scratch file, it was determined that a file with file name equal to unit number already existed.

1036 INFO\$ REQUEST FAILED

1037 STMT OPTION INVALID FOR HVTS ENVIRONMENT

One of the following occurred while processing an OPEN statement:

- The OFF clause was present.
- TYPE=ANSI, AREADA, APRNTA, APUNCH or APNCHA
- The record size specified in a Define File Block was greater than the default size.

1038 NOT ENOUGH TDFA SPACE OR FCT TOC FULL

In an HVTS environment, there is no room for another file in the TDFA or the maximum number of files are currently in use (the file control table of contents is full).

*1039 INQUIRE LITERAL TRUNCATED ON THE RIGHT

The literal being returned by the INQUIRE statement is longer than the receiving area and has been truncated on the right.

1040 INCORRECT NAMELIST SUBSCRIPT

- 1041 SIZE OF INTERNAL FILE EXCEEDED
- The formatted read of an internal file that is a character variable, character array element or character substring tried to read beyond the end of that file (see 5.9.1).
- 1042 INVALID STMT TYPE FIELD IN IO PACKET
- 1043 REREAD ILLEGAL FOR STMT TYPE OR SUBTYPE
- A reread unit may only be referenced by a formatted or list-directed READ statement.
- 1044 ATTEMPTED TO REREAD BEYOND END OF BUFFER
- The format used to reread the record tried to read beyond the reread record. The REREAD format may not contain slashes and cannot be shorter than the input/output list.
 - The list-directed reread tried to read more than one record.
- 1045 OPENED UNIT MAY NOT BECOME A REREAD UNIT
- If a unit is opened, it must first be closed before attempting to open it as a reread unit.
- 1046 INPUT BUFFER OVERWRITTEN
- The size of the record just read in exceeds the input buffer size. To create a larger input buffer use an OPEN or DEFINE FILE statement specifying the largest record size to be read.
- 1047 ERROR ON CLOSE, UNIT NOT OPEN FOR REREAD
- Execution of a CLOSE statement with the REREAD clause present has been attempted for a file opened for regular input/output.
- 1048 ERROR ON CLOSE, UNIT OPEN FOR REREAD
- Execution of a CLOSE statement without the REREAD clause present has been attempted for a unit opened for reread.
- 1049 TARGET ADDR USED IN MORE THAN ONE CLAUSE
- A variable or array element that receives a value in an INQUIRE statement may not be referenced by more than one of the clauses in the same INQUIRE statement.
- 1050 INVALID FILENAME FOR OPEN/INQUIRE
- Either an F-cycle, read/write keys, or an invalid character was encountered in a FILE clause of an OPEN or INQUIRE statement.

- 1051 A SCRATCH FILE MAY NOT BE KEPT
- *1052 RCD LARGER THAN MAX RCD SIZE TRUNCATED
- The size of the record just read via a symbiont read exceeds the maximum record size (default size if no OPEN or DEFINE FILE statement) for the file. The record will be truncated to the maximum record size.
- 1053 ATTEMPT TO READ A DUMMY RECORD
- A direct access read attempted to read a dummy record. The direct access record has not been written to yet and does not contain useful information. Dummy records are only detected in skeletonized files.
- 1057 ERROR ON TLBL\$ REQUEST
- An error was encountered when doing an ER TLBL\$. The substatus field of the I/O status word PTIOE will be set to the ER TLBL\$ error code. In addition, the ER TLBL\$ error code will be provided by a second line to the error message. The second line has the format:
- 'ER TLBL\$ ERROR CODE IS xxx'
- 1058 EQUIPMENT TYPE NOT ALLOWED
- Because of a PCIOS restriction, I/O operations are prohibited for word-addressable mass storage with equipment codes 020-027.
- 1059 INTERNAL FILE I/O IS INCORRECT
- An attempt was made to read an internal file using an empty FORMAT statement (FORMAT()) when the iolist was not empty.
- *1060 RECORD FORMAT ON OPEN IS INCONSISTENT
- At the opening of an existing direct access file, the record format specified in the OPEN or DEFINE FILE does not agree with the record format with which the direct access file was created.
- 1061 ERROR MSG NOT DEFINED
- 1062 RECURSIVE I/O CALLS NOT ALLOWED
- A function reference used in an input/output list attempted to perform I/O operations or referred to a subprogram which attempted to perform I/O operations. Note that this is a fatal error and control will not be returned to the program.
- 1063 DIRECT ACCESS BUFR TOO SMALL
- A larger buffer size for a direct access file must be specified in the OPEN statement. Refer to 5.10.1 for a discussion on the minimum buffer size.

G.10. FORTRAN DEFINE FILE Block Usage

The file description parameters given in the sequential OPEN or DEFINE FILE statements may also be specified in a Define File Block (DFB) which is external to the FORTRAN program. A DFB which is to be used by FORTRAN must exist in the file DFP\$. Prior to execution of the FORTRAN program, a DFB may be produced by the Define File Processor (DFP) as follows:

```
@DFP,E   fname .,DFP$.unit
```

where the E option indicates that DFB is to be produced. The variable *fname* specifies the external file name to which the DFB applies and is inserted in the DFB. DFP\$.*unit* is the file name and element name into which the DFB is inserted. For FORTRAN, this must be the file DFP\$ and the element name must be equivalent to a FORTRAN file reference number. The variable *fname* must be followed by a period or the default file TPF\$ will be associated with the unit number.

When the FORTRAN input/output routines attempt to retrieve a DFB, the file reference number from the sequential OPEN or DEFINE FILE statement is used as the element name of the DFB. The file DFP\$ must be assigned to the run by the user prior to execution of the FORTRAN program.

If a DFB is found in DFP\$, the input/output routines will perform the following @USE statement on the file, providing it is not a standard symbiont file:

```
@USE   x,q*f
```

where *x* is the unit number of the file and *q*f* is the qualifier and file name from the DFB. This *q*f* is the same as *fname* which was used by the DFP when the DFB was created. If the file is a buffered input/output file, the following assignment statement is also performed by the input/output routines:

```
@ASG   x.
```

If the file type from the OPEN or DEFINE FILE statement matches the file type from the DFB, the parameters which exist in OPEN or DEFINE FILE statements and DFB override the file default parameters respectively. If the file type from the OPEN or DEFINE FILE statement does not match the file type from the DFB, just the DFB parameters override the file default parameters.

The only DEFINE FILE statement or DFB parameter, other than *ft* (file type), which applies to symbiont files is *rs* (record size). Symbiont file unit members may be predefined in the file reference table (FRT). If a DEFINE FILE statement is executed for a predefined symbiont unit, the file type from the DEFINE FILE statement and the file type from the DFB (if a DFB exists) must match the file type in the FRT for the unit. If an OPEN or DEFINE FILE statement is executed for a symbiont unit which was not predefined in the FRT, the file type from the OPEN or DEFINE FILE statement, if not overridden by the DFB, is placed in the FRT and remains there for the duration of the run or until the unit is closed via the CLOSE statement and opened via the OPEN statement with a different file type.

Note that the Define File Processor may not be used to modify the parameters given in a direct access OPEN or DEFINE FILE statement, only sequential file parameters may be adjusted.

The following examples illustrate the DFP call and the keyword parameter lists applicable to the various FORTRAN file types.

- If *fname* is a sequential SDF file:

```
@DFP,E  fname.,DFP$.unit
FILE = SDF
RECORD = rs (record size)
BLOCK = bs (block size)
LINESIZE = ss (segment size)
```

where:

unit is the unit number
rs is a positive integer (optional)
bs is an integer greater than 224 (optional)
ss is a positive integer (optional)

- If *fname* is an ANSI file:

```
@DFP,E  fname.,DFP$.unit
FILE = ANSI
RECORD = rs / rf (record size/record format)
BLOCK = bs / bo (block size/buffer offset)
```

where:

unit is the unit number
rs is a positive integer (optional)
rf is U, F, V, FB, VB, VS or VBS (optional)
bs is a positive integer (optional)
bo is a positive integer less than *bs* (optional)

- If *fname* is a print symbiont file:

```
@DFP,E  fname.,DFP$.unit
FILE = ft (file type)
RECORD = rs (record size)
```

where:

unit is the unit number
ft is one of the following: APRINT, APRINT\$, APRNTA, APRNTA\$
rs is a positive integer less than or equal to 160

NOTE: The *fname* parameter is required by the DFP but is not used by the input/output routines for file assignment when the file type is for a standard symbiont file.

<u>Symbiont</u>	<u>Default Record Size in Chars</u>	<u>Allowable Range with DFP</u>
051 APRINT	132	$0 < rs \leq 160$
052 APUNCH	80	$0 < rs \leq 80$
053 AREAD	80	$0 < rs \leq 132$
054 APRNTA	132	$0 < rs \leq 160$
055 APNCHA	80	$0 < rs \leq 80$
056 AREADA	132	$0 < rs \leq 160$

For example, consider the following program called TEST:

```
COMMON  I,DATA(90)
DEFINE FILE 11(SDF,,10,224,112)
CALL SUB
DO 90  J = 1,1
90  WRITE(11,20) (DATA(K),K=1,J)
20  FORMAT (90(A1))
END
```

Assume that something is going wrong with writing unit number 11. The following DFP-modified execution would cause unit 11 output to go to the printer rather than to an SDF file:

```
@ASG,T  DFP$.
@DFP,E  DUM.,DFP$.11
FILE = APRINT
RECORD = 40
@XQT  TEST
```

Note that this use of DFP will change the maximum record size of all print units for this execution to 40 characters also.

G.11. Storage-Allocation Packet (Element M\$PKT\$)

A set of common storage allocation routines (which reside in the libraries of the ASCII FORTRAN, ASCII COBOL, and PL/I compilers) is used to control storage used by the I/O routines at run time (except in ASCII FORTRAN checkout mode). The packet used by the common routines is generated in the ASCII FORTRAN library as follows:

```
@PDP FTNPROC,FTNPROC
|MASM,S1 M$PKT$,M$PKT$
  SMS$PKT$PROC max
  END
```

where *max* is the maximum address that the program can reach (that is, common storage allocation will never request storage past this address). If *max* is -1 or 0, the default 0777777 (decimal value 262,143) is assumed.

The standard M\$PKT\$ is shipped with a default value for *max* (0777777).

The address specified for *max* must be sufficient to handle all I/O storage requests, or else error termination will occur.

The service routine MAXAD\$ (see 7.3.3.20.) may be called to change values in the common storage-allocation packet at run time.

Appendix H. Large Programs and the Multibanking Features of ASCII FORTRAN

H.1. Large Programs

The Series 1100 hardware architecture has a default 65,535 decimal word address range for the instructions of a collected program, including all user subprograms and all referenced run-time library routines. If the size of the collected program is larger than this 65K word range, the Collector will produce truncation errors because it is trying to place an address which is greater than 65K into a 16-bit instruction u-field. Index registers can hold an 18-bit address. Therefore, ASCII FORTRAN will generate code using index registers to hold addresses if the O option (the over-65K-address option) is used on the ASCII FORTRAN processor calls when the programs are compiled. This raises the boundary to a 262K word address range for a collected user program before truncation problems again appear.

If use of the O option results in no truncation errors during collection, the user's problem is solved. (Note that having the statement `COMPILER(PROGRAM=BIG)` in source programs is equivalent to using the O option when compiling them.)

However, if a program still gets truncation errors during collection, it is necessary to construct a multibanked program.

H.2. Banking

Banking is a Series 1100 mechanism for sharing the address space between different pieces of code or data. For example, the user can use the Collector `IBANK` directive to direct the Collector to construct a bank (an I-bank) holding subprogram X, and to construct another I-bank to hold subprogram Y. These two I-banks can be created with overlapping addresses. The same thing can be done with data. The user can use the Collector `DBANK` directive to place common block CB1 into one bank (a D-bank) and common block CB2 into another parallel D-bank using the same address space. These are called paged data banks. In this general manner, one creates a banked program which needs less address space than the unbanked program. Almost any number of I-banks and D-banks may be defined by the user, though the Collector documentation should be referenced for the exact limit (it is about 250). The user must construct a Collector symbolic (sometimes called a MAP symbolic) containing a sequence of Collector directives which define the banking structure of the user's program.

For this mechanism to work, generated code must be able to move an address window from bank to bank. A Processor State Register (PSR) is a hardware register which defines two address windows for an executing program, an I-bank window and a D-bank window. An LIJ (Load I-Bank Base and Jump) instruction moves the I-bank window, and an LDJ (Load D-Bank Base and Jump) instruction moves the D-bank window from one bank to another. Series 1100 hardware has two basic types, older single-PSR systems and newer dual-PSR systems. Single-PSR systems include the 1106, 1108, 1100/10, and 1100/20; dual-PSR systems include the 1110, 1100/40, 1100/60, and 1100/80. On dual-PSR systems the two PSRs are referred to as the Main PSR (PSRM) and the Utility PSR (PSRU). The 1100/60 and 1100/80 have the concept of four basing registers called Bank Descriptor Registers (BDRs). There is a one-for-one association between these four BDRs and the four windows as defined by the Main and Utility PSRs:

<u>BDR</u>	<u>PSR Window</u>
BDR0	I-bank PSRM
BDR1	I-bank PSRU
BDR2	D-bank PSRM
BDR3	D-bank PSRU

This appendix will refer to PSRs only since there is a one-for-one equivalent to BDRs.

Besides the LIJ and LDJ instructions to switch banks, the 1100/80 and 1100/60 have a generalized LBJ instruction which can be used to switch both I-banks and D-banks.

When constructing a Collector symbolic to create a multibanking structure for ASCII FORTRAN programs, certain conventions must be followed: (1) no address overlap should ever occur between I-banks and D-banks since results are unpredictable; (2) when multiple D-banks containing paged data are defined, they must start at the same address (FORTRAN's mechanism to switch D-bank basing depends upon this); (3) in any multibanked collection, one bank is defined as the control bank. The control bank is assumed to be always available and based since it will contain any unbanked programs and data and also the unbanked portions of the run-time library routines.

H.2.1. General Banking Example (Dual-PSR System)

The following example consists of three separately compiled elements. MAIN1 is the main program and SUB1 and SUB2 are subroutines. The first statement in each sample routine is a directive to the compiler to indicate that the final collected program will be banked, and appropriate linkages (e.g. LIJ, LDJ, LBJ instructions) must be used to ensure that the correct banks are visible when necessary. Note that the sizes of the code and data in the examples do not warrant the use of banking since these are simple examples for instruction only.

Example of a Main Program (MAIN1):

```
COMPILER (BANKED=ALL)
COMMON /cb1/ a,x /cb2/ b,y
CHARACTER*1 a,b
DATA a/'a'/, b/'b'/
WRITE(6,100) 'reference to:', a
WRITE(6,100) 'reference to:', b
a = 'c'
b = 'd'
WRITE(6,100) 'reference to:', a
WRITE(6,100) 'reference to:', b
100  FORMAT (1X, A13, 1X, A1)
x = 2.
y = sqrt(x)
z = x + y
WRITE(6,200) x, y
200  FORMAT (1X, 'sqrt of', F10.5, ' is', F10.5)
CALL sub1 (a, b, x, y, z)
END
```

Example 1 of a Subprogram (SUB1):

```
COMPILER(BANKED=ALL)
SUBROUTINE sub1 (a, b, x, y, z)
CHARACTER*1 a, b
WRITE(6,100) 'in subroutine sub1'
100  FORMAT(1X, A18)
WRITE(6,200) a, b, x, y, z
200  FORMAT(1X, 'a=', A1, ' b = ', A1, ' x = ', F10.5, ' y = ',
1F10.5, ' z = ', F10.5)
CALL sub2(b, 2.0)
END
```

Example 2 of a Subprogram (SUB2):

```
COMPILER (BANKED=ALL)
SUBROUTINE sub2(a, x)
CHARACTER*(*) a
CHARACTER*19 b3cell /'common block cb3'/
COMMON /cb3/ b3cell
CHARACTER*19 b4cell /'common block cb4'/
COMMON /cb4/ b4cell
PRINT *, 'a, len(a), x:', a, len(a), x
PRINT *, 'common block cb3:', b3cell
PRINT *, 'common block cb4:', b4cell
END
```

H.2.1.1. Collection of the General Banking Example

The following Collector symbolic could be used to collect the three sample program elements into a banked program. The LIB directive could be dropped if the ASCII FORTRAN library is in the system relocatable library, SYS\$*RLIB\$. This is the general form which should be followed for multibanking of ASCII FORTRAN programs on dual-PSR Series 1100 Systems.

```
LIB FTN*RLIB.
IBANK,MRD  IBANKM      .  INITIALLY BASED, MAIN PSR
      IN MAIN1
IBANK,RD   IBANK1,IBANKM
      IN SUB1
IBANK,RD   IBANK2,IBANKM
      IN SUB2
DBANK,UD   DBANK1,(O40000,IBANKM,IBANK1,IBANK2)
      .  INITIALLY BASED, UTILITY PSR
      IN F2ACTIV$(1)
      IN CB1
DBANK,D    DBANK2,DBANK1
      IN(MAIND) F2ACTIV$(3)
      IN CB2
DBANK,D    DBANK3,DBANK1
      IN(MAIND) F2ACTIV$(3)
      IN CB3
DBANK,CM   MAIND,(DBANK1,DBANK2,DBANK3)
      .  MAIND IS THE CONTROL BANK, ALWAYS BASED, ON MAIN PSR
      IN MAIN1
      IN SUB1
      IN SUB2
END
```

The collection of the sample program using this Collector symbolic would result in the banking structure shown in Figure H-1. The lines under the bank names are similar to the lines in a Collector S-option listing in that they indicate length.

If the three FORTRAN relocatables are copied to file TPF\$, and if the above Collector symbolic is in element TPF\$.BMAP, the user can collect this program into a banked absolute and execute it with the following control images. (This assumes that there are no other relocatables in file TPF\$.)

```
@MAP  BMAP ,BABS
@XQT  BABS
```

The following control images will do a default nonbanked collection of the program and will give the same results when executed.

```
@MAP , I MAP ,ABS
IN MAIN1
LIB FTN*RLIB.
@XQT  ABS
```


I-Banks Based on Main I-Bank PSR (PSRM) (start at 01000)	D-Banks Based on Utility D-Bank PSR (PSRU) (start at 040000 or over)	Control D-Bank Based on Main D-Bank PSR (PSRM) (starts after largest of D-banks based on utility D-bank PSR and may reach 262K limit)
IBANKM MAIN1 <hr/> IBANK1 SUB1 <hr/> IBANK2 SUB2 <hr/> C2F\$ I/O common bank <hr/>	DBANK1 CB1 <hr/> DBANK2 CB2 <hr/> DBANK3 CB3 <hr/>	Local data, library--MCORES\$ area <hr/>

- NOTES:**
1. Program code goes into I-banks.
 2. Named common blocks go into D-banks (paged data banks).
 3. Data local to subprograms, Blank Common, any programs or named common not placed into other banks and the run-time library routines, all go into the control bank.
 4. The C2F\$ I/O common bank is not mentioned in the collection, though it is referenced at run time for all I/O activities.
 5. The paged data banks must start at or after address 040000 to avoid address overlap with the hidden C2F\$ I/O common bank.
 6. The area after the Control bank is open for the I/O complex to acquire buffer space. Executive Requests (ERs) to MCORES\$ will be made at run time to expand this area.
 7. If any collected addresses go over 65K, the O option or the COMPILER (PROGRAM=BIG) statement should be used with all of the ASCII FORTRAN compilations.

Figure H-1. Dual-PSR Banking Structure

The execution of the banked or nonbanked absolute will result in the following output:

```
reference to: a
reference to: b
reference to: c
reference to: d
sqrt of 2.00000 is 1.41421
in subroutine sub1
a= c b = d x = 2.00000 y = 1.41421 z = 3.41421
a, len(a), x:d      1 2.0000000
common block cb3:common block cb3
common block cb4:common block cb4
```

H.2.1.2. Analysis of the Collector Symbolic

This subsection describes the Collector symbolic listed in H.2.1.

The main program MAIN1 is placed in an I-bank with the M option which makes it initially based on the main PSR. (The main program must be in an initially based bank.)

The other two routines, SUB1 and SUB2, are placed into two other I-banks, each starting at the same address as the I-bank containing MAIN1. (They could also be put into the same I-bank as MAIN1 since they are so small.)

All I-banks have the R option on the IBANK directive to indicate they are read-only I-banks. As a read-only bank, there will be less Executive swap file activity.

All D-banks have the D option on the DBANK directive to indicate that they are dynamic banks. This means that they may be swapped out by the Executive if they are not currently based, saving on main storage usage (though possibly causing more Executive swap file activity).

The bank names given on the IBANK and DBANK directives (for example, IBANKM) are called Bank Descriptor Indexes, or BDIs. The Collector gives them integer values which are used by the LIJ and LDJ bank-switching instructions.

The paged data banks contain named common blocks and must be based on the Utility D-bank PSR. The paged data bank DBANK1 was chosen to be the one initially based. The U option on the DBANK directive for DBANK1 indicates it is initially based on the Utility PSR. The other paged data banks holding named common are put at the same address as DBANK1.

The location counter one code (\$1 code) of the run-time "activate" element F2ACTIV\$ is put at the beginning of initially based paged data bank DBANK1. The same is done with \$3 code of F2ACTIV\$ for each of the other paged data banks. The F2ACTIV\$ \$1 code has the run-time code to do the hardware LDJ instruction to switch the paged data bank which is currently based. The \$3 code of F2ACTIV\$ is an exact copy of the \$1 code. They must be collected at the same address since this code is in control when the LDJ instruction switches address windows between paged data banks. The IN directive of F2ACTIV\$ (\$3) has MAIND in parenthesis. This is called local element inclusion and bypasses possible "LOCAL-GLOBAL CONFLICT" messages from the Collector.

The control bank MAIND is the D-bank named in the DBANK directive with the C option, and the M option on it means it is also initially based on the main D-bank PSR. Only the main program MAIN1 is included through use of an IN statement in this bank since the Collector will put anything not specifically included in another bank into the control bank. The control bank MAIND is placed after the largest of the three paged data banks so that no address overlap occurs.

The area after the end of the control bank is used by the storage management complex to obtain buffer space for the ASCII FORTRAN run-time system. Executive Requests (ERs) to M_{CORE}\$ are made to acquire this space.

Note that the three paged data banks must not be defined at less than an 040000 (octal) address and the paged data banks must start at an address higher than the highest I-bank address. This is because their address space would then overlap the C2F\$ I/O common bank and unpredictable results would occur.

Note that named common block CB4 was not given a home in any paged data bank. If the main program and any subprograms are explicitly included in an I-bank and a D-bank by an IN statement, CB4 will fall into the control bank and, since the control bank is always based, it is not being dynamically banked. This does not result in any problems; in fact, any subprograms not specifically included in a bank by an IN statement will fall harmlessly into the control bank. As long as the control bank does not get so large as to cause Collector truncation errors again, this is harmless. Any number of subprograms may be included in an I-bank, and any number of named common blocks may be included in a paged data bank. (Blank common and data local to subprograms must be in the control bank.) The criteria for the contents of a bank should be a function of the final collected size of the bank, and also a function of the locality of reference to the bank to try to minimize thrashing between banks. (Any problem caused by excessive Executive swap file activity can be minimized by carefully making selected banks static by not putting the D option on their I-bank or D-bank statements.) A reasonable size for an I-bank or paged data bank would be approximately 16,000 decimal words. This means that the control bank could be up to about 32,000 words in size before truncation problems again would occur.

H.2.1.3. Large Banks

If reasonable I-bank and paged data bank sizes still result in truncation errors at collection time, or, if large paged data banks (greater than approximately 30,000 words) are to be defined, the O option is needed on the ASCII FORTRAN processor calls to allow a 262K address range. In addition, the ordering of the paged data banks and the control bank must be inverted to prevent Collector truncation errors on the run-time library routines in the control bank. This means that the control bank must be placed after the I-banks in the address space, and before the paged data banks in the address space. Since the ASCII FORTRAN run-time system will make the control bank larger via ER M_{CORE}\$ to obtain buffer space, the user must leave enough room between the control bank and the paged data banks for I/O main storage requirements. (Appendix G contains formulas for estimating I/O main storage requirements for a program.) Allowing 10,000 decimal words is usually a sufficient amount. However, if an ER M_{CORE}\$ results in an address overlap of the control bank and paged data banks, error termination or a "hang" is ensured. The F2FCA library element may also be reassembled with a sufficiently large local area in it (see G.7).

To collect a user program with large D-banks, the Collector symbolic (from H.2.1.1) must be changed as follows. The DBANK directive for D-bank MAIND and the IN directive on MAIN1 are moved back to just before the DBANK1 definition. Then these statements are changed as follows:

```
DBANK,CM  MAIND
          IN MAIN1
DBANK,UD  DBANK1, (MAIND+10000)
```

The rest of the Collector symbolic remains unchanged.

The resulting Collector symbolic is:

```
LIB FTN*RLIB
IBANK,MR IBANKM . INITIALLY BASED, MAIN PSR
    IN MAIN1
IBANK,R IBANK1,IBANKM
    IN SUB1
IBANK,R IBANK2,IBANKM
    IN SUB2
DBANK,CM MAIND,(040000,IBANKM,IBANK1,IBANK2)
    .MAIND IS THE CONTROL BANK, ALWAYS BASED, ON MAIN PSR
    IN MAIN1
    IN SUB1
    IN SUB2
DBANK,UD DBANK1,(MAIND+10000) . INITIALLY BASED, UTILITY PSR
    IN F2ACTIV$( $1)
    IN CB1
DBANK,D DBANK2,DBANK1
    IN(MAIND) F2ACTIV$( $3)
    IN CB2
DBANK,D DBANK3,DBANK1
    IN(MAIND) F2ACTIV$( $3)
    IN CB3
END
```

The collection would then result in the banking structure of Figure H-2. The lines under the bank names are similar to the lines in a Collector S-option listing in that they indicate length.

H.2.1.4. Variations on the Dual-PSR Structure

The generalized example shows multiple I-banks and multiple D-banks being used at the same time. If the user's program is large only in the amount of I-bank code, the user can simply omit the definition of the paged data banks and only define I-banks and the control bank. If the program has large common blocks causing the size problem, then the Collector symbolic can be cut back to defining only one I-bank, the control bank, and multiple paged data banks.

The LIB directive tells the Collector where to obtain the ASCII FORTRAN run-time library. The simple form of the LIB directive causes all run-time library routines, both code and data, to fall into the control bank MAIND since they are not explicitly included in any bank. The following form of the LIB directive directs the Collector to put anything taken from FTN*RLIB into I-bank IBANKM, and D-bank MAIND, in a normal \$ODD/\$EVEN I-bank/D-bank split:

```
LIB FTN*RLIB.(IBANKM/$ODD,MAIND/$EVEN)
```

This can minimize the size of the MAIND control bank.

I-Banks Based on Main I-Bank PSR (PSRM) (starts at 01000)	Control Bank Based on Main D-Bank PSR (PSRM) (starts at 040000)	D-Banks Based on Utility D-Bank PSR (PSRU) (starts after the control bank and goes up to a 262K limit)
IBANKM MAIN1 <hr/> IBANK1 SUB1 <hr/> IBANK2 SUB2 <hr/> C2F\$ I/O common bank <hr/>	MAIND Local data, library-MCORE\$ area <hr/>	DBANK1 CB1 <hr/> DBANK2 CB2 <hr/> DBANK3 CB3 <hr/>

- NOTES:**
1. *The paged data banks may extend out to the 262K address limit.*
 2. *The 10K area between MAIND and the paged data banks is used by the I/O complex for buffers. If it is not sufficient, the separation must be increased or the library element F2FCA reassembled with a nonzero reserve.*
 3. *I/O acquires storage in increments of eight storage blocks (4096 decimal words).*

Figure H-2. Dual-PSR Banking Structure, Over 65K

H.2.2. Banking for Single-PSR 1100 Systems

The Collector symbolics described in H.2.1 through H.2.1.4 use the Utility PSR of dual-PSR systems to hold an address window for paged data. This Utility PSR does not exist on single-PSR systems such as the 1106, 1108, 1100/10, and 1100/20. If the user program needs only multiple I-banks and not multiple D-banks, the previously described Collector symbolics may be used with the definitions of the paged data banks removed. If the user program needs both multiple I-banks and multiple D-banks, it simply cannot be done on single-PSR hardware. However, the user can define a banking structure for multiple D-banks for single-PSR systems. A single I-bank is defined, and it is also made the control bank to hold all unbanked code and data. The paged data banks are defined to come after the control bank, but enough room must be left between them for I/O buffers (which are dynamically acquired by ER MCORES at run-time).

However, this type of collection has a problem resulting from the I-bank holding all unpagged data. Because of this, no common banks can be referred to at run-time to do I/O, calls to the Common Math Library (CML), etc. The run-time library used must be a very special one which has all run-time routines in relocatable form. (The ASCII FORTRAN library must be built as a type1 library, with the relocatable form of the PCIOS common I/O modules, and also the relocatable form of the CML modules.)

Another problem results from ASCII FORTRAN putting a SETMIN on all relocatables it generates which ensures that code to refer to array elements is correct. The Collector will emit a warning on each FORTRAN element. For example:

```
MAIN1 MINIMUM ADDRESS IGNORED-LCO NOT IN DBANK
```

These Collector warnings can be ignored, but the I-bank must be started at address 040000 or after to ensure that the FORTRAN generated code works correctly.

This special multiple D-bank banking structure should only be used on single-PSR systems. Its use, for example, on an 1100/80 could possibly result in incorrect results since the generated code assumes a different structure on the 1100/80, and LBJ instructions may be generated.

The example in H.2.1 using MAIN1, SUB1, and SUB2 could be collected using multiple D-banks for single-PSR systems with the following Collector symbolic:

```
LIB FTN*RLIBX.      . VERY SPECIAL LIBRARY !
IBANK,MC      IBANKM,040000      . CONTROL BANK NOW
  IN MAIN1, SUB1, SUB2
DBANK,MD      DBANK1, (IBANKM+10000)
  IN F2ACTIV$( $1)
  IN CB1
DBANK, D      DBANK2, DBANK1
  IN (IBANKM) F2ACTIV$( $3)
  IN CB2
DBANK, D      DBANK3, DBANK1
  IN (IBANKM) F2ACTIV$( $3)
  IN CB3
END
```

Collection would result in the following Collector diagnostics:

```

MAIN1  MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
SUB1   MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
SUB2   MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
CB4    MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
    
```

The collection would result in the banking structure given in Figure H-3 for the single-PSR multiple D-bank problem. The lines under the bank names in Figure H-3 are similar to the lines in a Collector S-option listing in that they indicate length.

One I-Bank (the Control Bank) Based on the Main I-Bank PSR (PSRM) (starts at 040000)	D-Banks Based on the Main D-Bank PSR (PSRM) (starts after the control bank and may go up to 262K)
IBANKM Code, library, unbanked data-MCORE\$ <hr/>	DBANK 1 CB 1 <hr/> DBANK 2 CB 2 <hr/> DBANK 3 CB 3 <hr/>

- NOTES:**
1. *There will be unavoidable Collector diagnostics.*
 2. *A totally local library must be used; no run-time common banks may be referenced.*
 3. *The 10K separation between IBANKM and the D-banks must be able to satisfy all buffer requests from the ASCII FORTRAN run-time system, or it must be increased, or the library element F2FCA must be reassembled with a nonzero reserve large enough to satisfy the buffer requests.*
 4. *The I-bank must start at or after address 040000.*
 5. *This setup should only be used on single-PSR systems requiring multiple D-banks. (It may possibly not operate on a SPERRY UNIVAC 1100/80 or an 1100/60 System.)*
 6. *If the paged data banks extend beyond 65K, the O option must be used on all of the ASCII FORTRAN compilations.*
 7. *IBANKM cannot extend beyond 65K in addressing.*

Figure H-3. Single-PSR Banking Structure

H.2.3. Banking, Efficiency, and Source Program Directives

The examples in H.2.1 through H.2.2 use a simple generalized directive to the ASCII FORTRAN compiler to indicate that banking will be used in the final absolute program. In fact, this generalized statement, `COMPILER(BANKED=ALL)`, means:

- Each subprogram referenced may be in a different I-bank, in the same I-bank, or the control bank.
- Input arguments may be in paged data banks, or in the control bank.
- Named common blocks may be in paged data banks, or in the control bank.

Therefore, the actual banking structure is virtually unknown to the compiler, and yet it must create linkages to ensure that items are visible or based when they are referenced.

H.2.3.1. I-Bank Linkages

The ASCII FORTRAN compiler uses the LIJ instruction to link between I-banks. This linkage is fairly efficient since the bank switch is done and then the called subprogram is entered, usually for some period of time. The compiler actually generates a pseudo-linkage called the IBJ\$ linkage. The Collector replaces this linkage with an LMJ instruction if the destination is in the control bank, or the same I-bank, and with an LIJ instruction if the destination is in a different I-bank.

H.2.3.2. D-Bank Linkages

Each time the compiler generates code to reference data in a (possibly) paged data bank (which may be currently based), it must also generate an "activate" sequence. This "activate" code has several variations, but a typical sequence is three instructions long.

Example:

Variables A, B, and C are in named common blocks CB1, CB2, and CB3.

The FORTRAN statement $A = B + C$ is to be compiled.

If the programmer has not given any directives to the ASCII FORTRAN compiler indicating that multiple D-banks are being used, three machine instructions will be generated for this statement: a LOAD, an ADD, and a STORE. If the programmer did indicate to the ASCII FORTRAN compiler that multiple D-banks are being used, there would also be three "activate" sequences generated. This results in 12 instructions instead of three for the unbanked program. In addition, each activate sequence links to the F2ACTIV\$ run-time routine which may execute up to 18 instructions if the desired bank is not currently based. One of these 18 instructions is an LDJ instruction. The LDJ instruction is simulated in the Executive for older single-PSR systems and may cause a presence-bit interrupt if the bank is swapped out on any Series 1100 System. Program efficiency is therefore extremely dependent upon the contents of the paged data banks and in the organization of the ASCII FORTRAN code.

Because performance can be so dramatically affected, several directives, including the BANK statement (see 6.6) and several COMPILER statement options (see 8.5), may be supplied to the ASCII FORTRAN compiler to help program efficiency.

H.2.3.3. Multiple I-Banks Only

If your collected program is constructed using multiple I-banks for code and does not define multiple paged data banks, then the LINK = IBJ\$ option of the COMPILER statement should be used rather than the more general BANKED=ALL option. The compiler will then generate the efficient IBJ\$ linkage for subprogram references and will generate code assuming that data is not banked. The resulting program should be as efficient as an unbanked program.

H.2.3.4. Multiple Paged Data Banks

Once the user has the multiple D-bank program debugged and running, the user might notice many "activate" code sequences in the generated code which are not necessary, since a given paged program bank may contain several named common blocks. Also, if the user is actually hopping between paged data banks a lot in the generated code, the user may wish for a much faster "activate" sequence.

H.2.3.4.1. The BANK Statement

The BANK statement associates a paged data bank BDI name with one or more named common blocks. Therefore, the user may tell the compiler that common blocks CB1, CB2, and CB3 are actually in the same paged D-bank and then the compiler will not generate activate sequences when the program references them. Also, since the BANK statement tells the compiler that these items are definitely banked and the BDI is supplied, a more efficient bank switch may be done. If the ASCII FORTRAN compilations are done on an 1100/60 or 1100/80, the compiler will generate a direct LBJ instruction to change which D-bank is currently based rather than calling the F2ACTIV\$ run-time routine. This is only generated on the 1100/60 and 1100/80 Systems since the LBJ instruction is simulated in the Executive on other Series 1100 Systems, and its operation is incompatible with the banking structure on single-PSR systems.

The BANK statement may be used alone, or in conjunction with the BANKED=ALL option of the COMPILER statement. However, if it is used alone, the compiler will assume that arguments are not banked, and that named common blocks not in BANK statements are not banked. If these items do happen to be banked, results will then be incorrect. So, it is safest to use the COMPILER(BANKED=ALL) statement in the FORTRAN programs and then supply some BANK statements after they are debugged to enhance efficiency.

If the single-PSR banking structure is to be used to collect the program, and it is compiled and executed on an 1100/60 or 1100/80, BANK statements cannot be used in the FORTRAN source. This is because the compiler generates code assuming the dual-PSR banking structure exists if the compilations are done on an 1100/60 or 1100/80 System and BANK statements are used. To develop a multi-D-banked program on the 1100/60 or 1100/80 to be run on any Series 1100 hardware, simply use the COMPILER (BANKED=ALL) statement in each program and do not use any BANK statements. Then it can be collected following the single-PSR method and be executed on any Series 1100 System.

Note that since BANK statements associate specific BDI names with common blocks, if a user changes the banking structure, the user must also change all the programs and recompile them.

The BANK statement and various COMPILER statement directives are described in 6.6 and 8.5.

H.2.3.4.2. Optimization and Program Organization

The ASCII FORTRAN compiler is only effective at remembering which bank is currently based and does not generate unnecessary activate sequences if global optimization is used during program compilation. (Global optimization is invoked with the Z option on the ASCII FORTRAN processor call.) Also, programmers should attempt to organize their code so that references to a given D-bank are grouped in areas of code. This is especially true for the inner loop of DO-loops. Users should try to have any inner loops refer to items in one paged data bank. (Unbanked data items can be referred to indiscriminately.)

Example:

```
SUBROUTINE SUBX (A, IA)
  COMPILER(BANKED=ALL)
  DIMENSION A(IA)
  COMMON/C1/A1(1000), B1(1000)
  COMMON/C2/A2(1000), B2(1000)
  BANK/BNK1/C1,C2
  COMMON WORK(1000) @BLANK COMMON
  IL = IA-1
  DO 10 I=1, IL
10   WORK(I)=A(I)/A(I+1)+.03
  DO 20 I=1, IL
    A1(I)=WORK(I)*B2(I)/B1(I)-A2(I)
    IF(A1(I).NE.0.0) A2(I)=1/A1(I)
20   CONTINUE
  END
```

The user has (possibly) banked arguments A and IA and two named common blocks which are known to be in D-bank BNK1.

Blank Common can never be banked, so the program does some initial processing on the input array A and moves it to WORK in Blank Common. (A local array could also have been used.) Since only one bank is being referenced inside the first loop, the "activate" code sequence will be moved out of the loop (if global optimization is used). The same thing is true for the second loop, and no activate sequences will be done inside the loop. Note that a single reference to an external routine inside either loop will cause at least one set of activate code to be generated inside the loop since the external routine could possibly change which paged data bank is currently based.

If the user had not supplied the BANK statement, the generated code would be loaded with activate sequences since the compiler must assume the worst case.

H.2.4. Banking Summary

- Programs constructed using multiple I-banks and no multiple D-banks should use the `COMPILER(LINK=IBJ$)` statement to indicate banking to the ASCII FORTRAN compiler.
- Programs which have multiple D-banks should use the `COMPILER(BANKED=ALL)` statement to indicate banking.
- Programs with multiple D-banks can cut the number of activate code sequences generated, and can cause the direct generation of LBJ instructions (1100/80 and 1100/60 only) by the selected use of BANK statements to associate named common blocks with specific paged data bank BDIs.
- If BANK statements are used in a FORTRAN program to enhance efficiency, no error diagnostics will occur if they are incorrect, and bad program results are ensured.
- The use of global optimization can cut the number of generated activate sequences dramatically.
- Judicious organization of program logic and careful definition of the contents of paged data banks can have a very beneficial influence on performance.
- There must never be an address overlap between any I-bank and any D-bank, or between the control D-bank and any paged data banks. If the run-time library used is a normal common bank, the C2F\$ I/O common bank can extend to address 037777 (octal). Therefore, no D-bank should start lower than address 040000.
- The control bank holds all unbanked user data and routines, all run-time library D-bank and any unbanked library routines. The control bank must be initially based and must never be unbased by an LIJ, LDJ, or LBJ instruction.
- If any addresses go beyond 65K in the collection, the O option (or the statement `COMPILER(PROGRAM=BIG)`) is needed on all ASCII FORTRAN compilations.
- Any element containing only Block Data Subprograms must be included via an IN directive in the control bank in the collection, since the Collector may otherwise ignore it. (This is true for nonbanked collections as well.)
- Control bank size can be somewhat minimized by supplying a LIB statement to the Collector that causes the code or I-bank portions of the run-time library to go into one of the user I-banks, for example, `LIB FTN*RLIB.(IBANKM/$ODD,MAIND/$EVEN)`.
- Multiple D-bank operation depends upon copies of the "activate" code existing in each paged data bank at exactly the same relative address. The \$1 and \$3 F2ACTIV\$ code segments are identical, and the only reason for the location counter split is to avoid local-global conflict messages during collection. The easiest way to ensure the same address for ACTIV\$ code is to make each paged data bank start at the same address and to have the ACTIV\$ code first in each D-bank.
- The local element inclusion of F2ACTIV\$ code is also important. The \$3 code (an exact copy of \$1 code) has no tags, but refers to data in the control bank; therefore, it must be visible only to the control bank.
- Single-PSR 1100 systems cannot have both multiple I-banks and multiple D-banks in the same collection.
- The single-PSR multiple D-bank setup needs a special library and cannot be used on an 1100/80 or 1100/60 if BANK statements are used in the ASCII FORTRAN source program. However, the setup may be used on any Series 1100 hardware if the `COMPILER(BANKED=ALL)` statement is used in the FORTRAN program and no BANK statements are used in that FORTRAN program, because LBJ instructions will *not* be generated.

Appendix I. Error Diagnostics in Checkout Mode

The diagnostics explained in Tables I-1 and I-2 are associated with the checkout mode of the FORTRAN (ASCII) compiler (see 10.6).

Table I-1. Messages Occurring During Program Load

Message	Explanation
**** CHECKOUT RELOCATION ERRORS ****	If any errors are encountered while loading the user program, this message will be issued and the error messages then printed.
WARNING: NAME IS UNDEFINED: <i>name</i>	The user has referenced a subprogram that is not defined in his compilation unit or the FTN library. The offending name is printed on the line following the message.
ERROR: USER PROGRAM TOO LARGE	An address generated while loading the user program will not fit in an address field. The user program is too large. It may fit if the O option is used, or if the Z option is omitted on the processor call card
NO MAIN ENTRY POINT, NO EXECUTION POSSIBLE	This message is produced if the user program does not contain a main program. Instead the program contains only subroutines, functions and BLOCK DATA subprograms. Interactive debug mode is entered.
BAD LINE NUMBER <i>n</i>	The line number, <i>n</i> (specified in the BREAK command), is either out of range for the user program or is on a nonexecutable statement.
BLOCK DATA PROGRAM NOT FOUND	The block data program specified in the <i>p</i> field of the PROG command or the <i>p</i> subfield of a command does not exist in the FORTRAN symbolic element.

Table I-2. Messages Generated by Interactive Debugging

Message	Explanation
COMMAND NOT ALLOWED	The GO command (no fields) or the WALKBACK command cannot be executed because normal execution of the FORTRAN program is not possible; that is, there is no main program in the element, or the CALL command has executed a subprogram and returned.
COMMAND NOT ALLOWED BECAUSE OF CONTINGENCY	The GO command (no fields) cannot be executed because a contingency has been captured by the compiler. For example, if the FORTRAN program encounters a guard mode (IGDM) contingency, then normal execution of the program cannot resume. Note that a RESTORE command may be used to bring back an original version of the program.
CONSTANT MUST BE TYPE <i>data-type</i>	The constant in the third field of the SET command is not the same data type as the variable in the first field. The SET command does not perform conversions between data types.
ELEMENT HAS NO MAIN PROGRAM	The main program is specified in the <i>p</i> field of the PROG command or in the <i>p</i> subfield of a command, but the FORTRAN symbolic element does not contain a main program.
ENTIRE ASSUMED-SIZE ARRAY CANNOT BE DUMPED	The range of an assumed-size array is not known. Only individual elements of an assumed-size array can be dumped.
ENTRY POINT NOT FOUND	The entry point specified in the <i>s</i> field of the CALL command or in the parameter list of the CALL command does not exist in the FORTRAN source.
ERROR: NO USER PROGRAM FOUND	The user is doing a RESTORE command but has not previously done a SAVE on the desired version.
FTEMP\$ STORAGE DESTROYED	A subprogram's temporary storage area (for saving registers and the parameter list) has been destroyed because of an error in the user program. The specified variable cannot be dumped.

(continued)

Table I-2. Messages Generated by Interactive Debugging (continued)

Message	Explanation
FUNCTION HAS NOT BEEN CALLED	A reference was made to a variable that is a character function entry point, but the function has not yet been called during execution of the FORTRAN program.
ILLEGAL COMMAND	An illegal debug command name was specified when input was solicited with ER ATREAD\$, or the name specified in the <i>cmd</i> field of the HELP command is not a debug command name.
ILLEGAL SYMBOLIC NAME	An illegal FORTRAN variable name was specified in the <i>v</i> subfield of a command, or an illegal subroutine or function name was specified in the <i>p</i> field of the PROG command or the <i>p</i> subfield of a command.
ILLEGAL SYNTAX	A general syntax error was found. This includes specifying a field for a command when none is required, or not specifying a field when one is required.
INCORRECT NUMBER OF SUBSCRIPTS FOR ARRAY *	The number of subscripts specified for the array in the <i>v</i> subfield of a command does not equal the number of dimensions declared for the array in the specified FORTRAN program unit.
IO ERROR ON LOADING USER PROGRAM, LOAD ABORTED	An I/O error occurred while accessing the user's program file during execution of the RESTORE command. The command is aborted. This may result in error termination also.
IO ERROR ON USER OUTPUT FILE, 'SAVE' COMMAND ABORTED	An I/O error occurred while accessing the user's program file during execution of the SAVE command. The command is aborted.
LABEL BREAK LIST IS FULL	An attempt was made to add an entry to the statement label break list with the command BREAK <i>n</i> L [/ <i>p</i>], but the list already has eight entries.
LABEL UNDEFINED *	The statement label <i>n</i> in the BREAK <i>n</i> L [/ <i>p</i>] command is not declared in the specified FORTRAN program unit.
LINE NUMBER BREAK LIST IS FULL	An attempt was made to add an entry to the line number break list with the command BREAK <i>n</i> , but the list already has eight entries.

(continued)

Table I-2. Messages Generated by Interactive Debugging (continued)

Message	Explanation
NO BREAK SET FOR LABEL *	The statement label n (in the specified program unit) in the command CLEAR n L [$/p$] is not in the statement label break list.
NO BREAK SET FOR LINE NUMBER	The line number n in the command CLEAR n is not in the line number break list.
PARAMETER'S SUBPROGRAM HAS NOT BEEN CALLED	An attempt was made to reference a subroutine or function parameter, but the subprogram has not yet been called during execution of the FORTRAN program.
PROGRAM UNIT NOT FOUND	The symbolic name specified in the p field of the PROG command or the p subfield of a command does not exist in the FORTRAN symbolic element as a named program unit.
SETBP NOT ALLOWED ON COMPILER GENERATED FOR SINGLE-PSR MACHINE	The SETBP command may only be executed on an ASCII FORTRAN compiler generated for a dual-PSR machine. The nonreentrant ASCII FORTRAN absolute taken off the test file (file 2) of the ASCII FORTRAN release tape is a compiler generated for 1108 (single-PSR).
SUBSCRIPT OUT OF RANGE FOR ARRAY	The constant subscripts specified for the array in subfield v of a command are too big or too small.
****UNDEFINED SUBROUTINE ENTRY****	During execution, the user program has called a function or subroutine which is undefined. The name of the subprogram was previously printed out with the Checkout relocation errors.
USER FILE REJECTED, NOT FASTRAND FORMATTED	A SAVE or RESTORE command is being attempted by the user, but the file is not a program file. Something has happened, making it unusable. The file affected is the relocatable output (RO) file specified on the @FTN processor call command.
USER INPUT FILE CANNOT BE ASSIGNED	The user's program file cannot be assigned to do the RESTORE command. Some other run must be using it.
USER OUTPUT FILE CANNOT BE ASSIGNED	The user's program file cannot be assigned to do the SAVE command. Some other run must be using it.

(continued)

Table I-2. Messages Generated by Interactive Debugging (continued)

Message	Explanation
VARIABLE IS AN ARRAY *	The variable in subfield <i>v</i> of a command has no subscripts, but the variable is declared as an array in the specified program unit. An array element is required.
VARIABLE IS NOT AN ARRAY *	The variable in subfield <i>v</i> of a command has subscripts, but the variable is not declared as an array in the specified program unit.
VARIABLE NOT DEFINED *	The variable in subfield <i>v</i> of a command is not declared in the specified FORTRAN program unit.
WARNING: CHARACTER CONSTANT TRUNCATED	The character constant in the third field of the SET command has too many characters to fit in the character variable. It has been truncated to the declared length of the variable.

* This error message is followed by a second printed line. This line specifies the program unit (in the FORTRAN element) from which the variable or statement label (in the command image) came. One of the following formats is used:

IN MAIN PROGRAM
IN MAIN PROG n

IN BLOCK DATA n
IN BLOCK DATA PROGRAM m

IN SUBROUTINE n[:e]

IN FUNCTION n[:e]

where:

n is a program unit name.

e is an external program unit name.

m is an unnamed block data program sequence number.

The specified program unit is taken from the *p* subfield of the command, or from the PROG command default program unit, if *p* is not specified in the command.

Appendix J. Comparison of ASCII FORTRAN Level 8R1 to Level 9R1 and Higher

J.1. General

ASCII FORTRAN levels 9R1 and 10R1 contain all the features of the FORTRAN standard, X3.9-1978 (called FORTRAN 77). ASCII FORTRAN level 8R1 does not have all these features. This appendix compares ASCII FORTRAN level 9R1 and higher to level 8R1. ASCII FORTRAN level 8R1 is missing the following six statements: PROGRAM (see 7.9), INTRINSIC (see 7.2.4), SAVE (see 7.12), OPEN (see 5.10.1), CLOSE (see 5.10.2), and INQUIRE (see 5.10.3). It is incompatible with ASCII FORTRAN level 9R1 and higher in the storage allocation of character data, DO-loops, the typing of parameter constants and statement functions, and list-directed input/output. It does not contain the 13 new intrinsic functions: ICHAR, CHAR, LEN, INDEX, ANINT, DNINT, NINT, IDNINT, DPROD, LGE, LGT, LLE, and LLT (see 7.3.1), nor the new logical operators, .EQV. and .NEQV.

A new option, STD=66, has been added to the COMPILER statement (see 8.5 and 8.5.5). This new option forces the ASCII FORTRAN compiler and library routines for level 9R1 and higher to execute as previous levels of ASCII FORTRAN did in the areas of storage of character data, DO-loops, typing of statement functions and parameter constants, and list-directed input and output.

This appendix is organized into subsections corresponding to the sections of the standard document, X3.9-1978. Each subsection of this appendix contains extensions in level 9R1 and higher over ASCII FORTRAN level 8R1 and conflicts between ASCII FORTRAN levels 9R1 and higher and level 8R1. ASCII FORTRAN extensions to level 8R1 in levels 9R1 and higher are features that have been implemented to make ASCII FORTRAN conform to the standard completely. Conflicts occur where the same construct can have different meanings in the two levels. Thus, conflicts imply that a change was made to a feature in ASCII FORTRAN level 8R1 to achieve compatibility with the standard.

J.2. FORTRAN Terms and Concepts

Extensions:

A main program may have a PROGRAM statement as its first statement. Level 8R1 has no PROGRAM statement (see 7.9).

Conflicts:

Character storage units for a datum are logically consecutive. Level 8R1 starts each character datum on a word boundary. The COMPILER statement provides an option, STD=66, to allow compatibility with previous levels on character data (see 8.5 and 8.5.5).

J.3. Characters, Lines, and Execution Sequence

Extensions:

PARAMETER statements may occur before and among IMPLICIT statements. Any specification statement that designates the type of a symbolic name of a constant must precede the PARAMETER statement that defines that particular symbolic name; the PARAMETER statement must precede all other statements containing the symbolic name of constants that are defined in the PARAMETER statement. Level 8R1 does not allow typing of PARAMETER constants (see 6.3 and 6.7). The COMPILER statement provides an option, STD=66, to allow compatibility with previous untyped PARAMETER constants.

Conflicts:

None.

J.4. Data Types and Constants

Extensions:

A complex constant may be written as a pair of integer constants or real constants. Level 8R1 allows only real constants (see 2.2.1.3).

Conflicts:

None.

J.5. Arrays and Substrings

Extensions:

Array names may be used in a SAVE statement. Level 8R1 does not have a SAVE statement (see 7.12).

Conflicts:

None.

J.6. Expressions

Extensions:

- Complex operands are allowed in relational expressions with the `.EQ.` and `.NE.` operators unless one operand is double precision. The comparison of a double precision value and a complex value is not permitted. Level 8R1 does not allow complex operands in relational expressions (see 2.2.3.3.1).
- The logical operators `.NEQV.` and `.EQV.` with lowest precedence are allowed. These operators are not in level 8R1 (see 2.2.3.3.1).

Conflicts:

None.

J.7. Executable and Nonexecutable Statement Classification

Extensions:

None.

Conflicts:

None.

J.8. Specification Statements

Extensions:

- EQUIVALENCE statements may contain character substring names. Level 8R1 does not allow character substring names in EQUIVALENCE statement lists (see 6.4).
- Integer constant expressions are allowed for subscript and substring expressions in EQUIVALENCE statements. Level 8R1 does not allow substring expressions in EQUIVALENCE statements (see 6.4).
- In the COMMON statement, an optional comma is allowed before the slash that comes before the common block name, that is, `[[,] / [cb] / nlist]`. No comma is allowed for level 8R1 (an error occurs) (see 6.5).
- A parameter constant may be typed via in an IMPLICIT statement or in an explicit type statement (see 6.3 and 6.7). Level 8R1 does not allow typing of parameter constants. The COMPILER statement provides an option, `STD=66`, to allow compatibility between levels 9R1 and higher and lower levels of ASCII FORTRAN for typing of parameter constants (see 8.5 and 8.5.5).

- The name of a statement function may appear in an explicit type statement. The name of a statement function may be typed by an IMPLICIT statement. Level 8R1 does not type statement functions (see 6.3 and 7.4.1). The COMPILER statement provides the option, STD=66, to allow compatibility between level 9R1 and lower levels of ASCII FORTRAN for typing of statement functions (see 8.5 and 8.5.5).
- The length in a CHARACTER type statement may be an asterisk or an integer constant expression in parentheses as well as just an unparenthesized constant (that is, *(*)* or *(exp)* or *const*). An entity in a CHARACTER statement must have an integer constant expression as a length specification unless that entity is an external function, a dummy argument of an external procedure, or a character constant that has a symbolic name. These exceptions may have a length specification of asterisk. The length specified for a character statement function or statement function dummy argument of type character must be an integer constant expression. Neither an asterisk nor an expression is allowed in the length specification in level 8R1 (see 6.3.2).
- The length for a CHARACTER array element may occur before and after the element (that is, *a(d)*length*). In level 8R1, the length may come before the subscript but an error is issued if it appears after the subscript (see 6.3.2).
- The comma is optional in the character type statement in the form:

```
CHARACTER[*len[,]] nam[,nam] . . .
```

Level 8R1 issues an error message for the comma following *len* (see 6.3.2).

- In the IMPLICIT statement, the length for character entities may be an unsigned, nonzero integer constant or an integer constant expression enclosed in parentheses that has a positive value. An unsigned, nonzero integer constant is allowed in level 8R1 but not an expression (see 6.3.1).
- A BLOCK DATA subprogram name may occur in an EXTERNAL statement. Level 8R1 does not allow the optional name for a BLOCK DATA subprogram (see 7.2.3 and 7.8.2).
- The INTRINSIC statement is used to identify a symbolic name as representing an intrinsic function. Level 8R1 does not have the INTRINSIC statement (see 7.2.4).
- The SAVE statement is used to retain the definition status of an entity after execution of a RETURN or an END statement in a subprogram. Level 8R1 does not have a SAVE statement (see 7.12).

Conflicts:

- Character equivalencing is based on character storage units in level 9R1 and higher and on words in level 8R1. This can give different results. This applies to both explicit EQUIVALENCE statements and to argument association and COMMON association. For example:

```
CHARACTER A*4,B*4,C(2)*3  
EQUIVALENCE (A,C(1)),(B,C(2))
```

Level 9R1:

1	2	3	4	5	6	7	8
A							
				B			
C(1)				C(2)			

Level 8R1:

1	2	3	4	5	6	7	8
A				B			
C(1)				C(2)			

The COMPILER statement provides an option, STD=66, to provide for compatibility with lower levels of ASCII FORTRAN (see 8.5 and 8.5.5).

- The PARAMETER statement is used to give a constant a symbolic name. If the type of the name is not default implied, the type must be specified by an explicit type statement or by an IMPLICIT statement prior to the appearance of the name in a PARAMETER statement. PARAMETER symbolic names have no type in level 8R1. Assignment of the value of the expression is done in level 9R1 and higher as in an assignment statement (that is, with type conversion, if necessary). The syntax of the PARAMETER statement is different in level 8R1 in that no parentheses may be used. Both forms of the PARAMETER statement syntax are allowed in level 9R1 and higher (see 6.7). The STD=66 option in the COMPILER statement provides for compatibility on previous levels of ASCII FORTRAN for typing of parameter constants (see 8.5 and 8.5.5).

J.9. DATA Statement

Extensions:

- The comma before the variable list is optional, that is, $[[,]nlist / clist /]$ The comma is not optional in level 8R1. A warning is given if the comma is omitted (see 6.8.1).
- Substring names are allowed in the variable list. Level 8R1 does not allow substring names in a DATA statement variable list (see 6.8.1).

Conflicts:

A PARAMETER constant beginning with the letter "O" in the constant list of a DATA statement will be interpreted by level 8R1 as an octal constant and by level 9R1 and higher as the PARAMETER constant (see 6.8.1).

For example:

```
PARAMETER (O2=5.)  
DATA A/O2/
```

J.10. Assignment Statements

Extensions:

None.

Conflicts:

None.

J.11. Control Statements

Extensions:

The DO-variable and the DO-statement parameters may be real or double precision as well as integer. Level 8R1 generates an error for noninteger DO-variables (see 4.5).

Conflicts:

In the standard, a DO-loop need not be executed. The iteration count is given by $\text{MAX}(\text{INT}((m2-m1+m3)/m3), 0)$. The DO-loop is not executed if $m1 > m2$ and $m3 > 0$ or if $m1 < m2$ and $m3 < 0$. In level 8R1, a DO-loop is always executed. The iteration count is given by $\text{MAX}(((e2-e1)/e3+1), 1)$. If $e1 > e2$ and $e3$ is omitted, level 8R1 assumes $e3 = -1$, and level 9R1 assumes $e3 = +1$ (see 4.5.4.1). The STD=66 option of the COMPILER statement provides for compatibility with previous levels of ASCII FORTRAN for DO-loops (see 8.5 and 8.5.5).

J.12. Input/Output Statements

Extensions:

- The internal unit identifier is a character variable or character array or character array element or character substring which specifies an internal file. Level 8R1 does not allow a character substring for an internal unit identifier (see 5.9.1 and 5.9.3).
- An empty I/O list is allowed on reads or writes to skip a record or to write an empty record. Level 8R1 compiler requires a nonempty I/O list on list-directed reads, unformatted sequential-access writes, list-directed write or print, and unformatted direct-access writes. Errors are issued when the list is missing (see 5.6.1.4, 5.6.2.2, 5.6.2.4, and 5.7.3).
- Character substring names are allowed in I/O lists. Level 8R1 only allows them in output lists (see 5.2.3).
- Character constants produced by list-directed output are not delimited by apostrophes, are not preceded or followed by a blank or comma, and do not have internal apostrophes represented by two apostrophes.
- The implied-DO list parameters are the same as the new DO-loop parameters (that is, more types, zero iterations possible). Level 8R1 does not allow the new parameters (see 4.5 and 5.2.3).
- The OPEN statement is not in level 8R1 (see 5.10.1).
- The CLOSE statement is not in level 8R1 (see 5.10.2).
- The INQUIRE statement is not in level 8R1 (see 5.10.3).

Conflicts:

None.

J.13. Format Specification

Extensions:

If the output format is $Gw.dEe$ and the value of the variable fits an F format, the format used will be $F(w-(e+2).d-i,(e+2)'b')$ where b is a blank. The format is $F(w-4).d-i,4('b')$ in level 8R1 (see 5.3.1).

Conflicts:

- During list-directed input, if the first record read in a read operation has no characters preceding the first value separator, this indicates a null field. In level 8R1 it does not indicate a null field but is handled the same as any other record. If the STD=66 option is used in the COMPILER statement, execution will choose level 8R1 and earlier methods of input.
- During list-directed output, character constants always have the PRINT format (that is, no apostrophes around character output). In level 8R1, PRINT and WRITE have different formatting in that apostrophes are used during the write. Also on list-directed output in level 9R1 and higher, a complex constant must be written on one record if it fits, by itself, on a record. Level 8R1 will break it up without checking to see if it fits on one line. If the STD=66 option of the COMPILER statement is used, execution will proceed with level 8R1 and earlier types of output.

J.14. Main Program

Extensions:

The PROGRAM statement to name a main program must be the first statement in the main program if it occurs. The PROGRAM statement is not implemented in level 8R1 (see 7.9).

Conflicts:

None.

J.15. Functions and Subroutines

Extensions:

- The following intrinsic functions are not in level 8R1: ICHAR, CHAR, LEN, INDEX, ANINT, DNINT, NINT, IDNINT, DPROD, LGE, LGT, LLE, LLT, UPPERC, LOWERC, and TRMLEN (see 7.3.1).
- Extended conversion intrinsic functions are the following: INT, REAL, DBLE, and CMLPX for all argument types. Level 8R1 gives warnings for use of complex with INT and proceeds to flag further uses of INT as a user function. FORTRAN 77 allows integer, real, complex, and double precision arguments for INT. Level 8R1 gives warnings for any use of REAL with variables other than complex and proceeds to flag further uses of REAL as a user function. FORTRAN 77 allows integer, real, and complex as arguments for REAL. Level 8R1 gives warnings for any use of DBLE with complex variables, and sets further calls of DBLE to a user function. FORTRAN 77 allows integer, real, double precision, and complex arguments for DBLE. Level 8R1 gives warnings for any complex variables used as arguments of CMLPX; it makes further uses of CMLPX become calls to a user function. FORTRAN 77 allows integer, real, double precision, and complex arguments for CMLPX (see 7.3.1).
- The FUNCTION statement has the length for a character function as CHARACTER[* length]C(A) while level 8R1 allows it after the function name, that is, CHARACTER FUNCTION C*3(A) (see 7.4.2.2). Both forms are allowed in level 9R1 and higher.
- Empty parentheses are allowed on the SUBROUTINE statement. The FORTRAN 77 form is SUBROUTINE sub [([d[,d]. . .])] while the level 8R1 form is SUBROUTINE sub [(d[,d]. . .)]. The forms sub and sub() are equivalent (see 7.4.3.2).
- An actual argument for a subroutine call may be *s, where s is a statement number. Level 8R1 uses currency signs (\$) or ampersands (&) for the statement number (see 7.2.1 and 7.2.2).
- A dummy argument array name may be associated with an actual argument which is an array element substring as well as an array or array element. Level 8R1 passes a temporary for an array element substring.
- A dummy argument which becomes defined may be associated with a substring as an actual argument. Level 8R1 associates it with a variable, an array element, or an expression.
- Empty parentheses are allowed in the ENTRY statement. The FORTRAN 77 form is ENTRY en [([d[,d]. . .])] while the level 8R1 form is ENTRY en [(d[,d]. . .)]. FORTRAN 77 requires that the function must be specified with the form en() even if the entry statement did not have the empty set of parentheses (see 7.7).

Conflicts:

- Statement functions are typeless in level 8R1, but are typed (just like other functions) in level 9R1 and higher. This can cause different results on account of implied type conversions (see 6.3 and 7.4.1). The STD=66 option of the COMPILER statement provides for compatibility of level 9R1 and higher with previous levels of ASCII FORTRAN for typing of statement functions (see 8.5 and 8.5.5).
- Register A1 contains a function packet address for character function references (see K.4.6). For level 8R1 ASCII FORTRAN, register A1 contains a result address for character function references. This is an incompatibility between level 8R1 and all higher levels. If a program compiled with level 9R1 or higher refers to a character function program compiled by level 8R1, the STD=66 option of the COMPILER statement must be present in the level 9R1 or higher program.
- If the value of e in RETURN[e] is less than one or greater than the number of asterisks in the subroutine entry, in level 9R1 control returns to the CALL statement that initiates the subprogram reference. In level 8R1, an error will be issued and the program will continue with unknown results (see 7.6).

J.16. Block Data Subprogram

Extensions:

An optional global name may be specified for a block data subprogram, that is, BLOCK DATA [*name*]. Level 8R1 does not allow the optional name.

Conflicts:

None.

Appendix K. Interlanguage Communication

K.1. ASCII FORTRAN (FTN) to SPERRY UNIVAC FORTRAN V

The FORTRAN V subprogram must be declared as:

```
EXTERNAL a (FOR)
```

or:

```
EXTERNAL *a
```

where *a* represents the FORTRAN V subprogram name.

The call to the FORTRAN V subprogram appears syntactically exactly as though it were a call to an ASCII FORTRAN subprogram.

Restrictions and Considerations:

- If both the ASCII FORTRAN and FORTRAN V programs perform I/O operations, the ASCII FORTRAN library element F2FCA must be reassembled. The allocation and releasing of buffer areas is not common between the two FORTRANs and I/O operations may fail. This problem is resolved by reassembling element F2FCA with the required amount of main storage. Refer to G.7 for determining the required amount.
- FORTRAN V Series E subprograms are restricted to symbiont types of I/O and the tag, CLOST\$, will be undefined at collection time. This can be ignored since CLOST\$ is defined in FORTRAN V Series T.
- Any files opened by FORTRAN V Series T must be explicitly closed via a CALL CLOSE statement in a FORTRAN V subprogram if they are to be usable after program termination.
- The FORTRAN V subprogram cannot call the EXIT service routine.
- The FORTRAN V subprogram name should not appear in a BANK statement.
- FORTRAN V subprogram arguments and function names should not be type character or double precision complex as FORTRAN V supports neither data type.
- The walkback mechanism will not work if a walkback is attempted from a FORTRAN V subprogram to an ASCII FORTRAN program.

K.2. ASCII FORTRAN to PL/I

The PL/I external procedure must be declared as:

```
EXTERNAL a(PL1)
```

where *a* represents the PL/I procedure name.

The call to the PL/I procedure appears syntactically exactly as though it is a call to a ASCII FORTRAN subprogram.

K.2.1. Restrictions and Considerations

- Level 8R1 PL/I, or later, must be used.
- Any file opened by a PL/I procedure must be closed by a PL/I procedure. Files may be shared between ASCII FORTRAN and PL/I, but they must be closed by the language which opened them before they can be accessed by the other language. A file that was opened by a given language does not have to be closed before switching to another language as long as the called language routine does not access the file.
- The PL/I procedure name cannot be a dummy argument name.
- An argument to a PL/I procedure cannot be a label, subprogram name or array name. An array element may be passed from ASCII FORTRAN to PL/I, but PL/I must declare its counterpart as a single data item, structures or cross-sections of arrays are not allowed.
- Passing an array name from ASCII FORTRAN to PL/I can be accomplished through common blocks. The PL/I procedure must use the EXTERNAL attribute on the declaration and the ASCII FORTRAN program must specify the array in a named common block. The PL/I common block name is the variable name with the EXTERNAL attribute. Note, however, ASCII FORTRAN stores arrays in column major order while PL/I stores them in row major order. This means that either the ASCII FORTRAN program must transpose the array so that it will be effectively in row major order when the PL/I procedure is called or the PL/I procedure must refer to the array with the subscripts in reverse order and the array dimensioned in reverse order.

For example:

<u>ASCII FORTRAN</u>	<u>PL/I</u>
EXTERNAL PL1SUB(PL1)	PL1SUB: PROC;
INTEGER ARR(2,4,6)	DCL 1 BLK1 EXTERNAL ALIGNED,
COMMON/BLK1/ARR	2 ARR(6,4,2) FIXED DECIMAL(10,0);
ARR(2,3,4) = 234	PUT SKIP ('WANT 234 :', ARR(4,3,2));
CALL PL1SUB	PUT SKIP;
END	END;

- IF execution is stopped by the PL/I procedure via the STOP statement, any files opened by ASCII FORTRAN will not be properly closed unless the ASCII FORTRAN program explicitly closes them via the CLOSE statement.

K.2.2. PL/I Argument Counterparts

PL/I has the following argument counterparts to ASCII FORTRAN.

ASCII FORTRAN	PL/I	Comment
INTEGER	FIXED BINARY (p,q) FIXED DECIMAL (p,q)	p can range 1-35, q must be 0. p can range 1-10, q must be 0.
REAL	FLOAT BINARY (p) FLOAT DECIMAL (p)	p can range 1-27. p can range 1-8.
DOUBLE PRECISION	FLOAT BINARY (p) FLOAT DECIMAL (p)	p can range 28-60. p can range 9-18.
COMPLEX	FLOAT BINARY COMPLEX (p) FLOAT DECIMAL COMPLEX (p)	p can range 1-27. p can range 1-8.
COMPLEX*16	FLOAT BINARY COMPLEX (p) FLOAT DECIMAL COMPLEX (p)	p can range 28-60. p can range 9-18.
LOGICAL	BIT (36) ALIGNED	Only the rightmost bit is used by ASCII FORTRAN. The PL/I string may not be of varying length.
CHARACTER* n	CHARACTER (n)	The PL/I string may not be of varying length.

K.3. ASCII FORTRAN to ASCII COBOL (ACOB)

The ASCII COBOL (ACOB) subprogram must be declared as:

```
EXTERNAL a(ACOB)
```

where *a* represents the ACOB subprogram name.

The call to the ACOB subprogram appears syntactically exactly as though it were a call to an ASCII FORTRAN subprogram.

Restrictions and Considerations:

- ACOB level 4R2, or higher, must be used.
- Any file opened by an ACOB subprogram must be closed by an ACOB subprogram. Files may be shared between ASCII FORTRAN and ACOB, but they must be closed by the language which opened them before they can be accessed by the other language. A file that was opened by a given language does not have to be closed before switching to another language as long as the called language routine does not access the file.
- The ACOB subprogram name cannot be a dummy argument name.
- It is the user's responsibility to ensure the data alignment is the same for an ASCII FORTRAN argument and its ACOB counterpart. Special care must be taken when passing character type data to ACOB. If the ASCII FORTRAN argument is not word-aligned, the ACOB argument declaration must reflect the offset via the use of a structure. To ensure that an ASCII FORTRAN character scalar or array is word-aligned, place it as the first item in COMMON or equivalence the character item to an integer variable.
- The ACOB subprogram name should not be a function name since ACOB does not support functions.
- An argument to an ACOB subprogram should not be a label or subprogram name since ACOB has no argument counterpart.
- An array name can be passed from ASCII FORTRAN to ACOB as an argument. However, ASCII FORTRAN stores arrays in column major order while ACOB stores them in row major order. This means either the ASCII FORTRAN program must transpose the array so that it will be effectively in row major order when the ACOB subprogram is called, or the ACOB procedure must reference the array with the subscripts in reverse order and the array dimensioned in reverse order. Beware of ASCII FORTRAN and ACOB alignment conventions.

Example:

<u>ASCII FORTRAN</u>	<u>ACOB</u>
EXTERNAL C(ACOB)	LINKAGE SECTION.
CHARACTER*5 ARR(2,4,6)	01 BUFF.
EQUIVALENCE (ARR(1,1,1),IDUM)	02 BUFA OCCURS 6 TIMES.
ARR(2,3,4) = 'ABCD'	03 BUFB OCCURS 4 TIMES.
CALL C(ARR)	04 BUFC PIC X(5) OCCURS 2 TIMES.
PRINT *, 'WANT EFG:', ARR(2,3,4)	PROCEDURE DIVISION USING BUFF.
END	C.
	DISPLAY 'WANT ABDC:',
	BUFC (4, 3, 2).
	MOVE 'EFG' TO BUFC (4, 3, 2).
	EXIT PROGRAM.

- If execution is stopped by the ACOB subprogram, any files opened by ASCII FORTRAN will not be properly closed unless the ASCII FORTRAN program explicitly closes them via the CLOSE statement.
- If ASCII COBOL passes a group item with no explicit type to an ASCII FORTRAN program, the corresponding ASCII FORTRAN argument can be type INTEGER, REAL, DOUBLE PRECISION or LOGICAL. CHARACTER type is not allowed. ASCII FORTRAN can only access that portion of a COBOL group item that is declared, either explicitly or implicitly, by the ASCII FORTRAN program.

K.3.1. ASCII COBOL Argument Counterparts

ASCII COBOL (ACOB) has the following argument counterparts to ASCII FORTRAN.

ASCII FORTRAN	ACOB	Comment
INTEGER	PIC S9(10) COMP SYNC	Ensure that the ACOB item is word-aligned.
REAL	COMP-1	
DOUBLE PRECISION	COMP-2	
COMPLEX	No ACOB counterpart	
LOGICAL	PIC 1 (36) SYNC	Only the rightmost bit is used by ASCII FORTRAN. Ensure that the ACOB item is word-aligned.
CHARACTER*(n)	PIC X(n)	Ensure that the alignment is the same for ASCII FORTRAN and ACOB.
TYPELESS †	PIC 1 (36) SYNC	Ensure that the ACOB item is word-aligned.

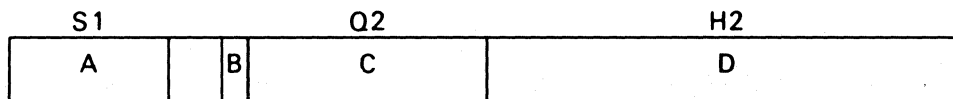
† Note that a typeless argument results from a typeless function, see 2.2.3.4.1.

K.4. ASCII FORTRAN and MASM Interfaces

This section provides information needed when writing Assembler routines which call or are called by ASCII FORTRAN routines.

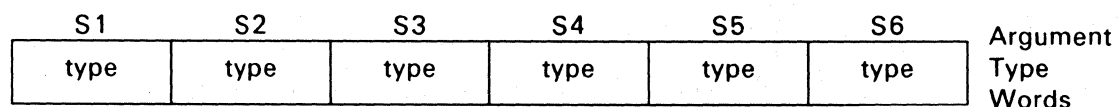
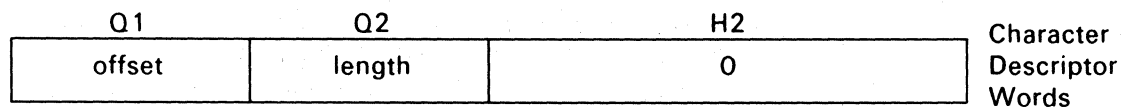
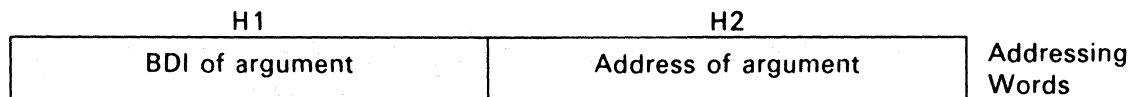
K.4.1. Arguments

For procedure calls with one or more arguments, ASCII FORTRAN requires an argument list. The address of the argument list is in H2 of register A0. A0 also contains the number of character arguments in S1 and the total number of arguments in Q2. Bit number 9 of A0 (leftmost bit is bit 1) specifies when argument type checking is desired by the caller. The following is the format of register A0 for calling an ASCII FORTRAN program.



where:

- A (S1 of A0) is the number of character arguments (maximum of 63).
- B (Bit 9 of A0, assume bits are numbered 1-36 and the leftmost bit is 1) is the argument type checking bit. If set to 1 by the caller, argument type checking will not be done. If set to 0, argument type checking will be done unless the called subprogram has disabled type checking. The called subprogram can disable type checking by using the COMPILER statement option ARGCHK=OFF or by compiling the called subprogram with optimization (Z or V option).
- C (Q2 of A0) is the total number of arguments (maximum is 150).
- D (H2 of A0) is the address of the argument list descriptor words that are explained in the following discussion.



The addressing words follow one another consecutively in storage. There is one addressing word for each argument. The bank descriptor index (BDI) is required if the ASCII FORTRAN subprogram being called expects banked arguments and the argument is not in the control D-bank. It is also required if the argument is an external subprogram name and the ASCII FORTRAN subprogram being called has the LINK=IBJ\$ or BANKED=ALL compiler statement options present. Otherwise, the BDI is zero. If an ASCII FORTRAN program calls a MASM routine with arguments that have a BDI associated with them, (that is, the actual data passed resides in a banked common block) the MASM routine is responsible for basing the argument's data bank. All D-bank basing must be done by an ASCII FORTRAN library routine (F2ACTIV\$). In other words, no LDJ or LBJ instructions should appear in user assembly code to switch the utility D-bank basing. The reason for this restriction is that the BDI of the currently based utility D-bank must always be contained in CURBDI\$ (a D-bank cell). The interface to the activate routine is:

```
LXI,H1  X11,0,A0  . place BDI in H1 of X11
LMJ     X11,ACTIV$. base D-bank
```

See Appendix H for details on ASCII FORTRAN and banked programs.

The character descriptor words follow one another consecutively in storage and follow the addressing words. A character function name or a Hollerith string passed as an argument does not have a character descriptor word. All other character types of arguments have a character descriptor word. The offset is the byte offset of the start of the character item within the word and has the value 0, 1, 2 or 3. If the character item begins on a word boundary, the offset is 0. A character item beginning on Q2 of the word has an offset of 1, an item beginning on Q3 has an offset of 2, and an item beginning on Q4 has an offset of 3. The character length, represented in number of characters, is in Q2 of the character descriptor word. If a character array is passed as an argument, the length passed is the size of an array element. If a character substring is passed as an argument, the length passed is the length of the substring.

If argument type checking is desired, bit 9 of A0 (assume bits numbered from 1 to 36 and the leftmost bit is 1) must be zero and the argument type words must be present. The argument type words follow one another consecutively in storage and follow the character descriptor words. There is one type word for each six arguments. The following is a list of allowable types:

- 0 Subprogram
- 1 Integer
- 2 Real
- 3 Double Precision Real
- 4 Complex
- 5 Double Precision Complex
- 6 Character
- 7 Logical
- 8 Label
- 9 Hollerith

Note that type 0, subprogram, will match any other type. Type 9, Hollerith, matches all types except character and label. All other types must match exactly or else a run-time diagnostic message is issued when type checking is enabled. Note the argument type words are optional. If they are not present, either the caller must specify so by setting bit 9 of A0 to 1 or the callee must disable type checking by using the COMPILER statement option ARGCHK=OFF or compiling with optimization.

If an Assembler routine is referring to an ASCII FORTRAN subprogram that has the COMPILER statement option STD=66 present or was compiled by an ASCII FORTRAN compiler lower than level 9R1, the contents of A0 and the packet format differ. S1 and bit 9 of A0 must be 0. The character descriptor words and the argument type words are not required. In addition, any character item passed to the ASCII FORTRAN subprogram must begin on a word boundary.

K.4.2. ASCII FORTRAN Register Usage

An ASCII FORTRAN subprogram saves and restores all registers that it uses except for the volatile set X11, A0-A5 and R1-R3. An ASCII FORTRAN subprogram requires register R15 to contain the address of a storage control table (F2SCT) which is used on I/O operations and by several of the ASCII FORTRAN library routines. Register R15 is loaded with the F2SCT address as part of the initialization performed by an ASCII FORTRAN main program. Once R15 has been initialized, it is the responsibility of any routine outside the ASCII FORTRAN environment to preserve its contents upon reentry to an ASCII FORTRAN subprogram.

K.4.3. Initializing the ASCII FORTRAN Environment

Under normal ASCII FORTRAN conditions, the ASCII FORTRAN environment is initialized by a call from the ASCII FORTRAN main program to the ASCII FORTRAN initialization routine. If an ASCII FORTRAN subprogram is called from an Assembler routine and there is not an ASCII FORTRAN main program, it is the responsibility of the Assembler routine to call the ASCII FORTRAN initialization routine. One of two ASCII FORTRAN initialization routines must be called by the Assembler routine before the ASCII FORTRAN subprogram is called. Note that the ASCII FORTRAN initialization need only be called once during the program execution. The initialization routines use only the volatile set of registers. Both initialization routines acquire buffer space and initialize tables that are used by I/O and ASCII FORTRAN library routines. This space is acquired by the Common Storage Management System. If the Assembler routine or controlling program has its own storage management system, the ASCII FORTRAN library element F2FCA can be modified and reassembled to avoid any ER MCORE\$. See Appendix G for details.

One of the initialization routines also registers a contingency routine for capturing contingency interrupts which is required for the proper execution of some ASCII FORTRAN programs. However, since some applications prefer to capture their own contingencies, a second routine which does not register contingencies is provided. Note that the ASCII FORTRAN service routines UNSET, OVSET and DIVSET will reregister contingencies and should not be called if an application depends on another contingency registration.

The following call will initialize the ASCII FORTRAN environment but will not register the ASCII FORTRAN contingency routine.

```
LMJ  A2,FINT$
```

The following call will initialize the ASCII FORTRAN environment and register the ASCII FORTRAN contingency routine.

```
LMJ  X11,FINT2$
```

Upon return from either of the initialization routines, register R15 contains the address of the ASCII FORTRAN Storage Control Table (F2SCT). It is the responsibility of the Assembler routine to ensure that R15 contains the F2SCT address when calling an ASCII FORTRAN subprogram.

K.4.4. Terminating the ASCII FORTRAN Environment

Under normal conditions, the ASCII FORTRAN environment is terminated by a call from the main program to the ASCII FORTRAN termination routine. The function of the termination routine is to output buffered I/O to the appropriate files and close all opened files. An ER EXIT\$ is then performed which terminates the program. If an Assembler routine calls an ASCII FORTRAN subprogram and control never reaches an ASCII FORTRAN main program for normal program termination, it is the responsibility of the Assembler routine to close all opened I/O files. The closing of files can best be accomplished by having the ASCII FORTRAN subprogram close them via the CLOSE statement. Note that if the ASCII FORTRAN termination routine is called, files will be closed but control is not returned to the caller. The following is the Assembler call to the ASCII FORTRAN termination routine.

```
LMJ  X11,FEXIT$
```

K.4.5. Calling an ASCII FORTRAN Subprogram

A call to an ASCII FORTRAN subprogram takes one of several forms depending upon whether the subprogram is banked or not. For a nonbanked ASCII FORTRAN subprogram call the following Assembler linkage can be used.

```
LXI,U  X11,0  
LMJ    X11,entry-point
```

ASCII FORTRAN will return to the caller via:

```
J      0,X11
```

For a banked ASCII FORTRAN subprogram call (that is, the ASCII FORTRAN subprogram has the compiler statement options BANKED=ALL, BANKED=RETURN, or LINK=IBJ\$), one of two calling sequences can be used:

```
LXI,U  X11,bdi-of-the-subprograms-bank  
LIJ    X11,entry-point
```

or:

```
LXI,U  X11,BDICALL$+entry-point  
IBJ$   X11,entry-point
```

For a description of IBJ\$ and BDICALL\$, see the Collector (MAP Processor) Programmer Reference, UP-8721.1 (see Preface).

An ASCII FORTRAN banked subprogram will return via:

```
LA,H1  A4,X11-save-location  
JZ     A4,0,X11  
LIJ    X11,0,X11
```

K.4.6. ASCII FORTRAN Function References

If an ASCII FORTRAN (level 9R1 or higher) character function is referred to, register A1 must contain:

H1	H2
0	<i>fcn-pkt-addr</i>

where *fcn-pkt* has the form:

H1		H2
0		<i>result-addr</i>
0	<i>char-length</i>	0

The *result-addr* field in H2 of the first word points to the caller's storage area where the result of the function will be stored. This storage area must be in the control bank or be visible to the function. It is the caller's responsibility to ensure that the function result area is large enough to hold the function result.

The *char-length* field in Q2 of the second word of the packet is the length of the function result expressed in number of characters.

For ASCII FORTRAN levels 8R1 and lower, and whenever the compiler statement option STD=66 is used in the function being called, register A1 must contain:

H1	H2
0	<i>result-addr</i>

The *result-addr* field description is the same as for levels 9R1 and higher. Note that a *fcn-pkt* is not required for levels lower than 9R1.

An ASCII FORTRAN character function will place the function result into the caller's storage area pointed to by *result-addr*. If the value of a function is not a character string, it is returned in registers A0, A0 through A1, or A0 through A3 depending on the function type.

K.4.7. Example

The following is an example of an Assembler routine that passes three arguments to the ASCII FORTRAN subroutine FTEST.

```

1.      AXR$
2.  $(1)
3.  MATH*  LMJ      A2,FINT$           . initialize FTN
4.
5.      L,U      R4,4                 . loop initialization count
6.  FTNREF LA      A0,(020003,ARGS)   . list pointer
7.      LXI,U    X11,0
8.      LMJ      X11,FTEST           . call nonbanked FTN rtn
9.      LA      A4,REALNO           . bump 2nd arg by 1.0
10.     FA      A4,(1.0)
11.     SA      A4,REALNO
12.     JGD     R4,FTNREF           . repeat call to FTN rtn
13.
14.     LMJ     X11,FEXIT$          . terminate program
15.
16.  $(0)
17.  CHARD  FORM    9,9,18
18.
19.  ARGS   +      STRING           . address of 1st arg
20.         +      REALNO          . address of 2nd arg
21.         +      STRING           . address of 3rd arg
22.     CHARD  0,3,0                 . 'ASM' offset=0 len=3
23.     CHARD  3,3,0                 . 'B17' offset=3 len=3
24.         +      6,2,6,0,0,0     . char real char type
25.
26.     ASCII
27.  STRING 'ASMB17'
28.  REALNO +      2.5
29.     END    MATH

```

The following is the ASCII FORTRAN subroutine FTEST that is called from the preceding Assembler routine:

```

1.      SUBROUTINE FTEST (CALTYP, X, CALID)
2.      CHARACTER CALTYP*(*), CALID*3
3.      *
4.      PRINT *, 'CALLER ID ', CALID, ' TYPE ', CALTYP
5.      PRINT *, 'SIN OF', X, ' = ', SIN(X)
6.      PRINT *, 'COS OF', X, ' = ', COS(X)
7.      RETURN
8.      END

```

The execution of the program will be:

```
CALLER ID B17 TYPE ASM
SIN OF 2.5000000 = .59847214
COS OF 2.5000000 = -.80114362
CALLER ID B17 TYPE ASM
SIN OF 3.5000000 = -.35078323
COS OF 3.5000000 = -.93645669
CALLER ID B17 TYPE ASM
SIN OF 4.5000000 = -.97753011
COS OF 4.5000000 = -.21079580
CALLER ID B17 TYPE ASM
SIN OF 5.5000000 = -.70554033
COS OF 5.5000000 = .70866977
CALLER ID B17 TYPE ASM
SIN OF 6.5000000 = .21511999
COS OF 6.5000000 = .97658762
```

At line 3 of the Assembler routine a call is made to initialize the ASCII FORTRAN environment. The ASCII FORTRAN initialization routine will acquire I/O buffer storage via the Common Storage Management System. If the Assembler routine has its own storage management system, the ASCII FORTRAN library element F2FCA would need modifications and reassembling. See Appendix G for details.

At line 6 register A0 is loaded with a literal that specifies two character arguments (S1), perform argument type checking (bit 9 of the literal is 0), and three total arguments (Q2). The second half of A0 contains the address of the argument list.

Lines 19 through 21 contain the argument addressing words. Lines 22 through 23 contain the character descriptor words for the first and third arguments, respectively, which are character types. The first argument passed is the character string "ASM", the second argument passed is the real number 2.5 (which is modified after each call by the Assembler routine) and the third argument is the character string 'B17'.

Line 24 contains the argument type word which the ASCII FORTRAN subprogram requires for argument type checking. The first and third arguments are character types as indicated by the 6 and the second argument is a real type as indicated by the 2.

At line 14 the ASCII FORTRAN termination routine is called which closes any opened I/O file and then terminates program execution via an ER EXIT\$.

Appendix L. ASCII FORTRAN Sort/Merge Interface

L.1. General

A Sort/Merge interface is available from ASCII FORTRAN to the Sort/Merge package. The Sort/Merge package is described in the Sort/Merge Programmer Reference, UP-7621 (see Preface).

L.2. Sort/Merge Features Available Through ASCII FORTRAN

The Sort/Merge interface is entered via the CALL statement in ASCII FORTRAN. The Sort/Merge interface provides the following functions:

- CALL FSORT Perform a sort.
- CALL FMERGE Perform a merge.
- CALL FSCOPY Specify an Assembler sort parameter table to be copied. This table may be used in subsequent sorts or merges. The use of such a table may also be inhibited.
- CALL FSSEQ Specify a user-specified collating sequence to be used in subsequent sorts or merges. The use of such a collating sequence may also be inhibited.
- CALL FSGIVE Deliver an input record to sort without leaving the user's input subroutine.
- CALL FSTAKE Receive an output record from sort without leaving the user's output subroutine.

L.3. Restrictions with Sort/Merge Interface

L.3.1. Banked Arguments Not Allowed

The data given to any area of the Sort/Merge interface must not be banked. The scratch area used by the Sort/Merge interface must not be banked.

L.3.2. Sort/Merge Interface Contains Only Formatted I/O

The Sort/Merge interface attempts only formatted I/O on all logical unit numbers used in the calls to do a sort or a merge. The ASCII FORTRAN level 10R1 I/O complex will not test the file to determine if the user does indeed have a formatted file. This could result in errors from the Sort/Merge interface.

L.3.3. Use of ASCII FORTRAN Free Core Area Element (F2FCA)

When the file R\$CORE is not assigned to the run (for FSORT only), the Sort/Merge interface will attempt to get storage space from the ASCII FORTRAN Library Common Storage Management System (CSMS). If the user has supplied a version of ASCII FORTRAN library element F2FCA so that the CSMS routines are not used, the user must make element F2FCA large enough to accommodate the storage area needed by the Sort/Merge interface routines, the Sort/Merge package, and the area needed for the FORTRAN library (see G.7).

L.4. The CALL Statement to FSORT

L.4.1. The CALL Statement for a Sort

The form of the CALL statement for a sort is:

```
CALL FSORT ( infost , inpt , outpt [ , comprt ] [ , datrd ] )
```

where:

infost is the information string, a character string that describes various parameters to the Sort/Merge interface, such as record sizes, key fields, and scratch facilities for FSORT. A key field (or a user's comparison routine) and a record size must be specified in *infost*.

Infost contains items of information separated by commas. Blanks in *infost* will be ignored. No distinction will be made between uppercase and lowercase alphabetic characters. *Infost* will be scanned from left to right and must be terminated by some character which is an illegal ASCII FORTRAN character, such as an exclamation point (!). An asterisk (*) must not be used to terminate *infost*.

Infost may contain several clauses. The mnemonics used will be truncated by the Sort/Merge interface to the first four characters. The following items may be used in *infost* for the call to FSORT:

1. RSZ=*rlch*

Rlch is the record length in ASCII characters. This record size must be specified when a sort is to be done. The RSZ clause may appear only once in *infost*. A record size clause may appear only once in *infost*; that is, the RSZ and VRSZ clauses may not appear in the same *infost*.

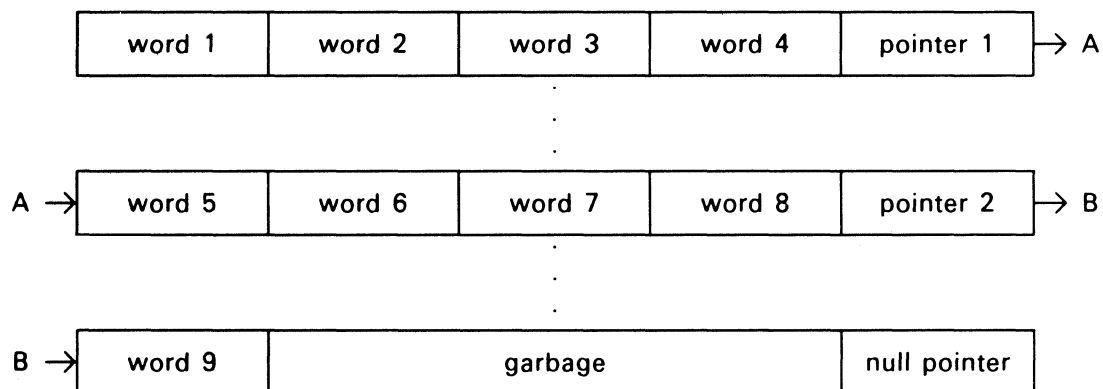
2. VRSZ=*mrlch* / *lnkszch*

Mrlch is the maximum record size in ASCII characters for variable length records. *Lnkszch* is an optional parameter indicating link size in ASCII characters. If *lnkszch* is omitted, the slash (/) may also be omitted. *Lnkszch* must be large enough to accommodate all keys. For example, if the keys are specified by:

```
KEY=(11/15,1/10/d/s)
```

the last character in any key field for this KEY specification is the 25th character. Therefore, the link size must be at least 25 characters long. The link size should be specified only when a comparison routine has been specified. The VRSZ clause may appear only once in *infost*. A record size clause may appear only once in *infost*; that is, the RSZ clause and the VRSZ clause may not both be used in the same *infost*.

When sorting variable length records, the records are separated into smaller parts (links) of equal size that are joined by pointers. As an example, consider a record of nine words with the link size four words. Schematically, the record would be stored as in the following figure:



When the link size is given in the VRSZ clause, the following rules should be considered:

1. The link size should not be too small. For example, if the link size is given as one word, the core (main storage) required for each record is exactly twice the record size. This means that the sort uses many more resources (main storage, mass storage, and tapes) than necessary.
2. The link size should not be too large, since this may mean that much of the area remains unused in the last link. This will cause problems because of poor use of main storage.

NOTE: *The two rules are in conflict. The choice of a link size requires a compromise between these two rules.*

It is frequently advantageous to sort short variable length records as if these records were fixed length records because of the resources used by the Sort/Merge package. If these records are treated as fixed length, the user must keep track of the record length.

3. CONS

CONS indicates that the closing messages from the Sort/Merge package are to be sent to the system console. If CONS is not present, the opening and closing messages from the Sort/Merge package are sent to the system log. The opening messages give the block sizes on mass storage and may be used to check the efficiency of the Sort/Merge usage of the scratch area. The closing messages give the input and output record counts and the bias of the data. The CONS specification may appear only once in *infost*.

4. DELL

DELL is used to indicate that the opening and closing messages from the Sort/Merge package should not be sent to the system log. This clause may appear only once in *infost*.

5. KEY= *keysp*

or:

KEY= *keyspn*

Keysp is a single key specification; *keyspn* is a multiple key specification of the form (*keysp*₁, *keysp*₂, . . .). The single key specification form, KEY= *keysp*, may occur a maximum of 40 times in *infost*. There may be a maximum of 40 *keysp* specifications within the *keyspn*. More than one KEY= *keyspn* clause may occur in *infost* but only 40 keys are allowed for each call to FSORT. Note that the limit of 40 keys includes any keys given in the sort parameter tables copied through the COPY clause (see L.7.1). The key specification may indicate a character key, that is, a key that begins and ends on a character boundary, or a bit key that either starts or ends outside a character boundary. The form of the character key is:

charpos / *length* / *seq* / *type*

where:

charpos is the position within the record of the most significant character of the key. Character positions are counted from left to right beginning with position 1.

length is an optional field that specifies the length of the key in characters. The default for *length* is one.

seq is an optional field that specifies the sequencing order of the key. The value A may be used for ascending order; D may be used for descending. The default value is A.

type is an optional field that specifies the type of the key. The value of this field may be B, Q, R, S, T, U, or V; U is the default value. The values for this field indicate:

B The key field contains a signed number in a Series 1100 System internal representation.

Q The key field contains a signed decimal number in 9-bit ISO character representation with a sign "overpunched" on the last digit.

R The key field contains 9-bit ISO characters with a leading sign, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, a minus, or a blank will be set to a blank.

S The key field contains 9-bit characters.

- T The key field contains 9-bit ISO characters with a sign in the last character position, that is, a plus, minus, or blank. Any character in the sign position that is neither a plus, minus, nor blank will be set to a blank.
- U The key field contains an unsigned number in the Series 1100 System internal representation.
- V The key field contains a signed decimal in 9-bit ISO characters with a sign "overpunched" on the first character.

The form of the bit key is:

BIT / wordpos / bitpos / length / seq / type

where:

wordpos is the position within the record of the word that contains the most significant bit of the record. Words within the record are numbered from 1.

bitpos is the position of the first bit of the key in the first word of that key. Bits are numbered from left to right beginning at 1.

seq is an optional field that specifies the sequencing order of the key: A for ascending or D for descending. The default is A.

type is an optional field that specifies the type of the key. The values for *type* may be A, B, D, G, L, M, P, or U; the default is U. The values for *type* indicate:

- A The key field contains 6-bit characters. All A key fields must start and end on 6-bit byte boundaries.
- B The key field contains a signed number in the Series 1100 System internal representation.
- D The key field contains 6-bit Fielddata characters with a leading sign, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank will be set to a blank. All D key fields must start and end on 6-bit byte boundaries.
- G The key field contains 6-bit Fielddata characters with a sign in the last character, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank will be set to a blank. The key field must begin and end on a 6-bit byte boundary.
- L The key field contains a number in 6-bit Fielddata characters with a sign "overpunched" on the first digit. The key field must start and end on a 6-bit byte boundary.
- M The key field contains a number in signed magnitude representation. This means that the first bit is the sign (that is, a 1 for negative and a 0 for positive), and the rest of the field is the absolute value of the number.

- P The key field contains a signed decimal number in 6-bit Fieldata characters with a sign "overpunched" on the last digit. The key field must begin and end on a 6-bit byte boundary.
- U The key field contains an unsigned number in Series 1100 System internal representation.

6. COMP

COMP is used to indicate the use of a user comparison routine. This indicates the presence of *comp* in the call to FSORT. The COMP clause may appear only once in *infost*.

7. COPY

COPY is used to indicate that an Assembler sort parameter table is to be copied. The COPY clause may appear only once in *infost*. See L.6.1.

8. DATA

DATA is used to indicate that a user data reduction routine is present. This indicates the presence of *datrd* in the call to FSORT. This option may only be specified when fixed length records are to be sorted. The DATA clause may appear only once in *infost*.

9. SELE=*recno1*

or:

SELE=*recno1* / *recno2*

or:

SELE=(*recno1* [/ *recno2*], *recno3* [/ *recno4*], . . .)

Recno1 through *recno4* are record numbers. The SELE, or select, clause is used to indicate which records are to be given to the Sort/Merge package. If the first form is used, only the record specified by *recno1* will be given to the sort. If the second form is used with *recno2*, all records from *recno1* through *recno2* will be given to the sort. If the third form of the SELE clause is used, the records between each pair of record numbers will be given to the sort and single records will be given to the sort. All records will be read, but only those records specified in the SELE clause will be given to the sort. Only 10 record number pairs can be used in the third form. For each pair, *recno1* must be less than or equal to *recno2*, and the last number of each pair must be less than the first number of the next pair. If *recno1* appears without *recno2*, or *recno3* appears without *recno4*, only *recno1* or *recno3*, respectively, will be given to the sort. This clause may appear only once in *infost*.

10. CORE=*corsz*

Corsz is the size in words of the scratch area to be used by the sort. At least 3000 words must be used. In general, the sort runs faster if the scratch area given to sort is expanded. This clause may appear only once in *infost*. See L.9.2.

11. FILE= *file-name*

or:

FILE=(*file-name* , *file-name* , . . .)

File-name is a Series 1100 System internal file name. The second form of the FILE clause permits the specification of more than one file name within the clause. See L.9.3.1. The following restrictions apply to scratch files:

1. All scratch files must be assigned when the sort starts executing.
2. A maximum of 26 scratch files may be specified.
3. At least three tape scratch files must be used if any tape scratch files are used.
4. If tape scratch files are used, a maximum of two mass storage scratch files may be used for the sort.

12. NOCH= *chksm*

Chksm is any combination of the letters D, F, K, and T. The letter T refers to tape and D, F, and K refer to mass storage. The nocheck clause is used to omit a checksum. If the sort uses one or two mass storage scratch files, D refers to the smaller of the two (one must be at least twice the size of the other) and F refers to the larger of the two, if both files are present.

If the sort uses three or more mass storage scratch files, K refers to the checksum on all the files.

If K is specified for a sort with fewer than three mass storage files, D and F are assumed. If D or F is specified for a sort with more than two mass storage scratch files, K is assumed. This clause may appear only once in *infost*. See L.9.3.2.

13. MESH= *meshsz* / *device*

Meshsz is the mesh size and *device* is any combination of the letters D, F, K, and T. If *meshsz* is not given, the value 5 is assumed. If *device* is not present, the mesh size is assumed to apply to all device types. The letter T indicates the use of tape scratch files; D, F, and K indicate the use of mass storage scratch files. D and F are used if one or two mass storage scratch files are used. D refers to the smaller of the two mass storage scratch files and F refers to the larger of these two files. The letter K indicates the checksum of three or more mass storage scratch files. The MESH= specification may appear only once in *infost*. The letters D, F, K, and T may be used only once each in the MESH specification. See L.9.3.2.

14. BIAS= *biasno*

Biasno is the average number of records in sorted subsequences present in the input file. For example, *biasno* will be 1 if the input is in exactly reverse order. For random data, *biasno* is 2. If the input file is in an almost sorted order, the bias value will be higher. The BIAS clause may appear only once in *infost*. Giving the bias value, if known, may improve the performance of the sort substantially. See L.9.1.

An example of an information string *infost* to sort variable length records with a maximum length of 200 characters with four key fields would be the following:

```
'VRSZ=200,KEY=(1/10//s,11/10/d/q,21/10,31/10/d),CONS!'
```

The key fields in the record are defined as follows:

1. The first key starts in the first character position of the record, is 10 characters long, and is to be sorted in ascending order with a user-specified collating sequence (if that sequence is present).
2. The second key starts in character position 11, is 10 characters long, and is to be sorted in descending order with an overpunch in the last character position.
3. The third key starts in character position 21, has a length of 10 characters, and is to be sorted in ascending order.
4. The fourth key starts in character position 31, has a length of 10 characters, and is to be sorted in descending order.

The CONS clause is present so that all messages will be sent from the Sort/Merge package to the console.

If one must sort record images of 80 characters with a key that starts in character position 1, that has a length of 5 characters, and that is sorted in ascending order, the information string *infost* may be:

```
'RSZ=80,KEY=1/5!'
```

Infost must be the first parameter in the call to FSORT. The other parameters follow *infost*.

inpt is either a logical unit number or the name of an input subroutine. If *inpt* is the name of an input subroutine, the subroutine must be declared in an EXTERNAL statement in the program unit containing the call to FSORT. See L.8.1.

outpt is either a logical unit number or the name of an output subroutine. If *outpt* is the name of an output subroutine, the subroutine must be declared in an EXTERNAL statement in the program unit containing the call to FSORT. See L.8.4.

comprt is the name of a comparison subroutine supplied by the user. The subroutine is called whenever two records are to be compared. The name of the comparison subroutine must be declared in an EXTERNAL statement. This parameter must not be present if the user has not provided a comparison subroutine. This parameter must be present if the COMP clause occurs in *infost*. See L.8.2.

datred is the name of a data reduction subroutine. The name of *datred* must be declared in an EXTERNAL statement. This subroutine will be called whenever two records with equal keys have been found. It will decide whether the two records are to be merged into one record or not merged. This feature is used for sorting fixed length records only. *Datred* must not be present if the user has not specified the DATA clause in *infost*; it must be present if the DATA clause occurs in *infost*. See L.8.3.

L.4.2. Examples of Sort with Logical Unit Numbers

The following runstream contains a call to FSORT with a simple information string *infost* which contains a KEY clause and a RSZ clause. The RSZ clause gives a record size of 80 characters. The KEY clause states that the key begins in the first character position of the record, has a length of five characters, and will be sorted in ascending order. *Infost* ends with an exclamation point (!). The input and output parameters are simply unit numbers 5 and 6. The Sort/Merge interface does formatted reads on unit 5 until all the input data has been read. The interface then calls the Sort/Merge package to sort the data, and does formatted writes on unit 6 of the data from the Sort/Merge package.

```
@RUN
@FTN,SI

      CALL FSORT('key=1/5,RSZ=80!',5,6)
      END

@MAP,SIF
LIB  ASCII*FTNLIB.
@XQT
... data images to be sorted ...
@FIN
```

Another example of a simple sort is given in the following runstream. This program assumes that the source input from file IN*PUT will be written to the file OUT*PUT. The records are 80 characters long with keys starting in character positions 1 and 6. Each key is 5 characters long. The first key is sorted in ascending order, and the second key in descending order.

```
@RUN
@FTN,SI

      CALL FSORT('rsz=80,key=(1/5,6/5/d)!',9,10)
      END

@MAP,SIF
LIB  ASCII*FTNLIB.
@ASG,A IN*PUT
@ASG,C OUT*PUT
@USE 9,IN*PUT
@USE 10,OUT*PUT
@XQT
@FIN
```

L.4.3. Examples of Sort with User Subroutines

The following example is a simple variation of the first example in L.4.2. The RSZ clause declares the record size to be 80 characters. The KEY clause indicates that the key starts in the first character position of the record, has a length of 5 characters, and will be sorted in ascending order. The *inpt* and *outpt* parameters are user-supplied input and output subroutines which are declared in an EXTERNAL statement in the program. The subroutines contain formatted I/O statements to read from unit 5 and write to unit 6.

```
@RUN
@FTN,SI

      EXTERNAL IN,OUT
      CALL FSORT(key= 1/5,rsz= 80!,IN,OUT)
      END

@FTN,SI  IN

      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER*4 RECORD(20)
      READ(5,1,END=2) RECORD
      LENGTH=80
      IEOF=0
      RETURN
1     FORMAT(20A4)
2     IEOF= 1
      RETURN
      END

@FTN,SI  OUT

      SUBROUTINE OUT(RECORD,LENGTH)
      CHARACTER*4 RECORD(20)
      IF (LENGTH.GE.0) WRITE(6,1) RECORD
1     FORMAT(1X,20A4)
      RETURN
      END

@MAP,SIF
LIB ASCII*FTNLIB.
@XQT
... data images to be sorted ...
@FIN
```

Another example of a sort with user I/O routines is given in the following runstream.

```
@RUN
@FTN,SI
```

```
EXTERNAL IN,OUT
CALL FSORT('rsz=80,key=(1/5,6/5/d),core=20000',IN,OUT)
END
```

```
@FTN,SI IN
```

```
      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER*4 RECORD(20)
      READ(9,1,END=2) RECORD
1     FORMAT(20A4)
      LENGTH=80
      IEOF=0
      RETURN
2     IEOF=1
      RETURN
      END
```

```
@FTN,SI OUT
```

```
      SUBROUTINE OUT(RECORD,LENGTH)
      CHARACTER*4 RECORD(20)
      IF (LENGTH.LT.0) GO TO 2
      WRITE(10,1) RECORD
1     FORMAT(20A4)
      RETURN
2     ENDFILE 10
      RETURN
      END
```

```
@MAP,SIF
LIB ASCII*FTNLIB.
@ASG,A IN*PUT
@ASG,C OUT*PUT
@USE 9,IN*PUT
@USE 10,OUT*PUT
@XQT
@FIN
```

L.5. The CALL Statement to FMERGE

L.5.1. The CALL Statement for a Merge

The form of the CALL statement for a merge is:

```
CALL FMERGE (infost, inpts, outpt [, comprt ])
```

where:

infost is the information string, a character string that describes various parameters to the Sort/Merge interface, such as record sizes, key fields, and scratch facilities for FMERGE. A key field (or a user's comparison routine) must be specified in *infost*.

Infost contains items of information separated by commas. Blanks in *infost* will be ignored. No distinction will be made between uppercase and lowercase alphabetic characters. *Infost* will be scanned from left to right and *infost* must be terminated by some character which is an illegal ASCII FORTRAN character, such as an exclamation point (!). An asterisk (*) must not be used to terminate *infost*.

Infost may contain several clauses. The mnemonics used will be truncated by the Sort/Merge interface to the first four characters. The following items may be used in *infost* for the call to FMERGE:

1. RSZ=*rlch*

Rlch is the record length in ASCII characters. This record is not required. If the RSZ clause is not given, the Sort/Merge interface will assume that the maximum record length is 1000 words. This will waste some main storage. If the RSZ clause is present, the interface will check that the records from each input source are in sequence. The RSZ clause may appear only once in *infost*. Only one record size clause may appear in *infost* at a time. Thus, the VRSZ clause may not be used if the RSZ clause is used in *infost*.

2. VRSZ=*mrlch* / *lnkszch*

Mrlch is the maximum record size in ASCII characters for variable length records. *Lnkszch* is an optional parameter indicating link size in ASCII characters. If *lnkszch* is omitted, the slash (/) is also omitted. *Lnkszch* must be large enough to accommodate all keys. For example, if the keys are specified by:

```
KEY=(11/15,1/10/d/s)
```

the last character in any key field for this KEY specification is the 25th character. Therefore, the link size must be at least 25 characters long. The link size should be specified only when a comparison routine has been specified. The Sort/Merge will check to see if the keys fit in the link size but will otherwise ignore the link size for a merge. The VRSZ clause may appear only once in *infost* and only one record size clause may be specified in *infost* at a time. Thus, the RSZ clause may not be used if the VRSZ clause appears in *infost*.

3. KEY= *keysp*

or:

KEY=(*keyspn*)

Keysp is a single key specification and *keyspn* is a multiple key specification of the form *keysp*₁, *keysp*₂, etc. The single key specification form KEY= *keysp* may occur a maximum of 40 times in *infost*. There may be a maximum of 40 *keysp* specifications within *keyspn*. More than one KEY= *keyspn* clause may occur in *infost*, but only 40 keys are allowed for each call to FMERGE, including any keys copied through the COPY clause (see L.7.1). The key specification may indicate a character key, that is, a key that begins and ends on a character boundary, or a bit key that either starts or ends outside a character boundary. The form of the character key is:

charpos / length / seq / type

where:

charpos is the position within the record of the most significant character of the key. Character positions are counted from left to right beginning with position 1.

length is an optional field that specifies the length of the key in characters. The default for the length is one.

seq is an optional field that specifies the sequencing order of this key: A for ascending and D for descending. The default value is A.

type is an optional field that specifies the type of the key. The value of this field may be B, Q, R, S, T, U, or V; the default is U. The values for this field indicate:

B The key field contains a signed number in a Series 1100 System internal representation.

Q The key field contains a signed decimal number in 9-bit ISO character representation with a sign "overpunched" on the last digit.

R The key field contains 9-bit ISO characters with a leading sign, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank will be set to a blank.

S The key field contains 9-bit characters.

T The key field contains 9-bit ISO characters with a sign in the last character position, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank will be set to a blank.

U The key field contains an unsigned number in a Series 1100 System internal representation.

V The key field contains a signed decimal in 9-bit ISO characters with a sign "overpunched" on the first character.

The form of the bit key is:

BIT/*wordpos* / *bitpos* / *length* / *seq* / *type*

where:

wordpos is the position within the record of the word that contains the most significant bit of the record. Words within the record are numbered from 1.

bitpos is the position of the first bit of the key in the first word of that key. Bits are numbered from left to right beginning at 1.

seq is an optional field that specifies the sequencing order of the key: A for ascending or D for descending. The default is A.

type is an optional field that specifies the type of the key. The values for *type* may be A, B, D, G, L, M, P, or U; the default is U. The values for *type* have the following meanings:

- A The key field contains 6-bit characters. All A key fields must start and end on 6-bit byte boundaries.
- B The key field contains a signed number in a Series 1100 System internal representation.
- D The key field contains 6-bit Fielddata characters with a leading sign, that is, a plus, minus, or blank. Any character in the sign position that is neither a plus, minus, nor blank is set to a blank. All D key fields must start and end on 6-bit byte boundaries.
- G The key field contains 6-bit Fielddata characters with a sign in the last character, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank. The key field must begin and end on a 6-bit byte boundary.
- L The key field contains a number in 6-bit Fielddata characters with a sign "overpunched" on the first digit. The key field must start and end on a 6-bit byte boundary.
- M The key field contains a number in signed magnitude representation. This means that the first bit is the sign, that is, a 1 for negative and a 0 for positive, and the rest of the field is the absolute value of the number.
- P The key field contains a signed decimal number in 6-bit Fielddata characters with a sign "overpunched" on the last digit. The key field must begin and end on a 6-bit byte boundary.
- U The key field contains an unsigned number in Series 1100 System internal representation.

4. COMP

COMP is used to indicate the use of a user comparison routine. This requires the presence of *comprt* in the call to FMERGE. The COMP clause may appear only once in *infost*.

5. COPY

COPY is used to indicate that an Assembler sort parameter table is to be copied. This clause may appear only once in *infost*. See L.6.1.

6. INPU=*inptsor*

Inptsor is an integer constant from 2 through 24 that indicates how many input sources are given in the parameter *inpts*. The INPU clause must appear only once in *infost*.

An example of an information string *infost* that merges two files containing variable length records with a maximum length of 200 characters with four key fields is:

```
'VRSZ=200,KEY=(1/10//s,11/10/d/q,21/10,31/10/d), INPUT=2!'
```

The key fields in the record are defined as follows:

1. The first key starts in the first character position of the record, is 10 characters long, and is sorted in ascending order with a user-specified collating sequence if that sequence is present.
2. The second key starts in character position 11, is 10 characters long, and is sorted in descending order with an "overpunched" sign in the last character position.
3. The third key starts in character position 21, has a length of 10 characters, and is sorted in ascending order.
4. The fourth key starts in character position 31, has a length of 10 characters, and is sorted in descending order.

To merge three files containing records of 80 characters with a key that starts in character position 1, that has a length of 5 characters, and is sorted in ascending order, the information string *infost* may be:

```
'RSZ=80, INPUT=3, KEY=1/5!'
```

Infost must be the first parameter in the call to FMERGE. The other parameters follow *infost*.

inpts is two or more logical unit numbers, the names of two or more input subroutines, or a combination of logical unit numbers and input subroutines. The parameter *inpts* may contain from 2 through 24 input sources. If *inpts* contains the names of input subroutines, the subroutine names must be declared in an EXTERNAL statement in the program unit containing the call to FMERGE. See L.8.1.

outpt is either a logical unit number or the name of an output subroutine. If *outpt* is the name of an output subroutine, the subroutine must be declared in an EXTERNAL statement in the program unit containing the call to FMERGE. See L.8.4.

comprt is the name of a comparison subroutine supplied by the user. The subroutine is called whenever two records are to be compared. The name of the comparison subroutine must be declared in an EXTERNAL statement. This parameter must not be present if the user has not provided a comparison subroutine. This parameter must be present if the COMP clause is used in *infost*. See L.8.2.

L.5.2. Examples of CALL Statements to Merge

The following runstream contains a call to FMERGE with a simple information string *infost* which contains a KEY clause and a RSZ clause. The RSZ clause declares a record size of 80 characters. The KEY clause indicates that the key starts in the first character position of the record, has a length of 5 characters, and will be sorted in ascending order. *Infost* contains the clause INPUT=2 to indicate that there are two input sources contained in the input parameter *inpts*. The *inpts* parameters are the logical unit number 8 and the user-supplied input subroutine name IN. The output parameter *outpt* is the logical unit number 9. *Infost* ends with an exclamation point (!).

```
@RUN
@FTN,SI  MAIN
```

```
EXTERNAL IN
CALL FMERGE('rsz=80,key= 1/5,input=2!',8,IN,9)
END
```

```
@FTN,SI  IN
```

```
SUBROUTINE IN(RECORD,LENGTH,IEOF)
CHARACTER*4 RECORD(20)
READ(5,1,END=2) RECORD
LENGTH=80
IEOF=0
RETURN
1  FORMAT(20A4)
2  IEOF=1
RETURN
END
```

```
@MAP,SIF
LIB ASCII*FTNLIB.
```

```
@ASG,A IN
@ASG,C OUT
@USE 8,IN
@USE 9,OUT
@XQT
```

```
... the second input file on data images ...
```

```
@FIN
```


The example that follows will merge two input image files (IN*1 and IN*2) that have been sorted in ascending order on columns 1 through 10 and the merged data will be written to file OUT*PUT. The input subroutine will ignore all records in the input file IN*2 that have a 1 in column 11.

```
@RUN
@FTN,SI

      EXTERNAL IN
      CALL FMERGE('rsz=80,key=1/10,input=2!',8,IN,10)
      END

@FTN,SI  IN

      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER RECORD*80,I,ONE/'1'/
1      FORMAT(A)
2      FORMAT(10X,A1)
3      READ(9,1,END=4) RECORD
      DECODE(2,RECORD) I
      IF (I.EQ.ONE) GO TO 3
      LENGTH=80
      IEOF=0
      RETURN
4      IEOF=1
      END

@MAP,SIF
LIB ASCII*FTNLIB.
@ASG,A IN*1
@ASG,A IN*2
@ASG,C OUT*PUT
@USE 8,IN*1
@USE 9,IN*2
@USE 10,OUT*PUT
@XQT
@FIN
```

L.6. The CALL Statement to FSCOPY

L.6.1. The CALL Statement to Copy an External Sort Parameter Table

This facility is provided primarily to provide access to the Assembler procedure R\$FILE.

The form of the CALL statement to copy an external Assembler sort parameter table is:

```
CALL FSCOPY( table )
```

where *table* is the name of an externalized entry point. *Table* must be declared in an EXTERNAL statement. The CALL statement to FSCOPY with one external argument must occur before a call to FSORT or FMERGE with the COPY clause in its information string *infost*. (See L.4.1 and L.5.1.) The call of FSCOPY establishes which sort parameter table is to be copied. The subroutine argument is the first word of the sort parameter table to be copied. The subroutine argument is not an ASCII FORTRAN subroutine but is an Assembler entry point.

Only one sort parameter table may be copied at any one time. Each call on FSCOPY deletes the previous table that was copied. If FSCOPY is called without any arguments, a new table will not be copied and the previous table will be deleted.

L.6.2. Record Size When FSCOPY Is Used

Key positions, record lengths, and link sizes in Assembler sort parameters must be given as if there were an extra word in front of the record. (See L.4.1 and L.5.1.)

L.6.3. An Example of CALL Statement to FSCOPY

The following runstream contains an Assembler sort parameter table and program which calls FSORT and FSCOPY. The program will sort the source input from character positions 1 through 6 in ascending order, from character positions 7 through 12 in descending order, and from character positions 13 through 16 in ascending Fielddata order with a special collating sequence such that all B's precede all A's. Note that the information for character positions 13 through 16 came from the Assembler sort parameter table. The source input is on logical unit 5 and the output is placed on logical unit number 6. Note the extra word or six characters in the starting character position (19+6). This is described in L.6.2.

```
@RUN
@MASM,SI   COPIED
           R$FILE 'KEY',19+6,6,'A','A' ; Extra word!
COPIED*    'SEQ','@','UPTO',' ',' ;
           'B','A','ALL'
           END

@FTN,SI

           INTEGER CORE(21000)
           EXTERNAL COPIED
           CALL FSCOPY(COPIED)
           CALL FSORT('key=(1/6,7/6/d),copy,rsz=80,core=20000!',
1           5,6,CORE)
           END

@MAP,IFS
LIB ASCII*FTNLIB.
@XQT
. . . data images . . .
@FIN
```

L.7. The CALL Statement to FSSEQ

L.7.1. The CALL Statement to Provide a User-Specified Collating Sequence

A user-specified collating sequence may not be explicitly declared in the information string *infost* of a call to FSORT or FMERGE. The use of a nonstandard collating sequence is specified only in the KEY clause field *type* in a character key with the value S. (See L.4.1 and L.5.1.) The user-specified collating sequence will be set up through a call to FSSEQ with a single argument.

The form of a CALL statement to FSSEQ is:

```
CALL FSSEQ ( seqtbl )
```

where *seqtbl* is an argument containing a character string that is 256 characters long. *Seqtbl* contains the user-defined collating sequence of the ISO character set. If *seqtbl* is not present, the previous user-defined collating sequence is deleted. Only one user-defined ISO collating sequence will be in use at any one time. A second CALL statement to FSSEQ will cause the previous collating sequence to be replaced with the new user-defined collating sequence.

L.7.2. An Example of the CALL Statement to FSSEQ

The following runstream contains two calls to FSORT and two calls to FSSEQ. The first call to FSSEQ contains a user-defined collating sequence in array SEQTAB. The collating sequence is the same as the standard ISO collating sequence except that the letters A and B (uppercase and lowercase) are interchanged. The first call to FSORT uses the user-defined collating sequence. The input is read from unit 5. The input is sorted according to:

1. The first key that starts in character position 1 of the record, that has a length of 10 characters, and that is sorted in ascending order.
2. The second key that starts in character position 11 of the record, that has a length of 5 characters, and that is sorted in descending order according to the user-defined collating sequence specified in the call to FSSEQ.
3. The third key that starts in character position 16 of the record, has a length of 5 characters, and is sorted in ascending order.

The result is written by the user output routine OUT.

The second call to FSSEQ deletes the previous user-defined collating sequence and does not set up another sequence. This means that the normal ISO collating sequence will be used when sorting. Note the use of the CORE= clause in *infost* in both calls to FSORT.

```
@RUN
@FTN,SI

EXTERNAL OUT
INTEGER SEQTAB(64),CHAR
* set up collating sequence
  CHAR(I)=BITS(SEQTAB(1+I/4),1+9*MOD(I,4),9)
DO 1 I=0,255
1   CHAR(I)=I
  CHAR(65)=CHAR(65)+1
  CHAR(66)=CHAR(66)-1
  CHAR(97)=CHAR(97)+1
  CHAR(98)=CHAR(98)-1
* give collating sequence to sort
  CALL FSSEQ(SEQTAB)
* do the first sort
  CALL FSORT('key=(1/10,11/5/d/s,16/5),rsz=80,
1   core=20000!',5,OUT)
* remove collating sequence
  CALL FSSEQ
* do the second sort
  CALL FSORT('key=(1/10,11/5/d/s,16/5),rsz=80,
1   core=20000!',10,6)
  END

@FTN,SI  OUT

SUBROUTINE OUT(RECORD,LENGTH)
CHARACTER*80 RECORD
IF (LENGTH.LT.0) RETURN
WRITE(10,1) RECORD
PRINT 2,RECORD
RETURN
1  FORMAT(A)
2  FORMAT(1X,A)
END

@MAP,SIF
LIB ASCII*FTNLIB.
@ASG,T TEMP
@USE 10,TEMP
@XQT
. . . data images . . .
@FIN
```

L.8. User-Specified Subroutines

The Sort/Merge interface allows the user to provide subroutines to do the following:

- Read records
- Compare records
- Examine fixed length records with equal keys and optionally merge the records
- Write records

The user is not required to supply any of these subroutines. The Sort/Merge interface and package will handle all these areas if the user does not wish to supply any subroutines.

L.8.1. User-Specified Input Subroutine

An input subroutine may be supplied to be called by the Sort/Merge package to read the records. (See L.4.1 and L.5.1.) The input subroutine will be called with three arguments. The first argument is an array which will contain the input record to be returned to the Sort/Merge package. The second argument is an integer that contains the length of the input record in characters. The third argument is an integer that indicates when the last record has been delivered.

The input subroutine may do the following:

1. Read a record and return the record to the sort.
2. Return the null string with a record length of zero and the third argument set to a one to indicate the end of the input file.
3. Read a record and call FSGIVE with that record as an argument. The input subroutine may enter FSGIVE several times before returning an input record or an end-of-file mark to the Sort/Merge package.

The end of the file can be signaled through FSGIVE. Control will not be returned to the instruction following the call to FSGIVE in the input subroutine. Control will be returned to the Sort/Merge package.

L.8.1.1. An Example of a User-Specified Input Subroutine

The following input subroutine will read the source input from unit 5 and return a record length of 80 characters. The third argument is set to zero if the end of the file is not reached and set to one if the end of the file is reached in the input file.

```
SUBROUTINE IN(RECORD,LENGTH,IEOF)
CHARACTER*4 RECORD(20)
READ(5,1,END=2) RECORD
LENGTH=80
IEOF=0
RETURN
1  FORMAT(20A4)
2  IEOF=1
RETURN
END
```

L.8.1.2. The CALL Statement to FSGIVE

The call to FSGIVE provides the capability of giving a record to the sort without leaving the user-specified input subroutine. FSGIVE will be called with three arguments. These arguments are similar to the arguments for the input subroutine. The first argument is the input record for sort. The second argument is the length of the record given to the sort. The third argument is the flag given to sort to indicate that the end of the file has been reached. If the flag is zero, the end of the file has not been reached, while a nonzero flag indicates that the end of the file has been found.

L.8.1.3. An Example of User-Specified Input Subroutine with FSGIVE

The following input subroutine will read characters separated into words by blanks or the end of the line from input unit 5. Any character except a space may be part of a word.

The input subroutine will read from unit 5 when first entered. The subroutine will move each word that it has found to the record area and then call FSGIVE for each word that is not the last word on a data image. The input subroutine IN will be reentered each time a new data image is needed from the input file. The end of the input file is signaled by the input subroutine IN. The end of the input file could also have been indicated by setting the third argument to FSGIVE to a nonzero value. The calls to FSGIVE will be intermixed with calls to the input subroutine IN.

```
      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER RECORD*80,CARD(80),BLANK/ ' ' /
1     READ(5,2,END=99) CARD
2     FORMAT(80A1)
*     find last nonblank character
      DO 3 IMAX=80,1,-1
3     IF (CARD(IMAX).NE.BLANK) GO TO 4
*     The input record was blank, so read a new record
      GO TO 1
4     I=1
*     find first blank separator
      DO 6 J=1,IMAX
6     IF (CARD(J).EQ.BLANK) GO TO 8
*     record has no more blanks - deliver
      ENCODE(80,2,RECORD) (CARD(J),J=1,IMAX)
7     LENGTH=80
      IEOF=0
      RETURN
*     At least one blank was found
8     IF (J.NE.1) GO TO 9
*     It was a leading blank - ignore it
      I=I+1
      GO TO 5
*     A word was found - deliver
9     ENCODE(80,2,RECORD) (CARD(K),K=I,J-1)
      CALL FSGIVE(RECORD,80,0)
      GO TO 5
*     This is end of input - tell the sort
99    IEOF=1
      RETURN
      END
```

L.8.2. A User Comparison Routine

If a user comparison routine is present, it will be called whenever the Sort/Merge package must compare two records. The COMP clause must be present in the information string of the call to FSORT or FMERGE. The parameter *comprt* must also be specified in the call to FSORT or FMERGE. (See L.4.1 and L.5.1.)

The compare subroutine will be called with three arguments. The first two arguments are the two records to be compared when the records are fixed length records or the first links of the two records to be compared when the records are variable length. The third argument is an integer whose value informs the sort of the result of the comparison done by the comparison subroutine. The result may be:

1. The value 1 if the first record precedes the second record.
2. The value 2 if the order of the records is immaterial.
3. The value 3 if the second record precedes the first record.

Care is necessary when using a comparison subroutine together with keys specified in the information string because the Sort/Merge package translates the key fields according to certain rules. The Sort/Merge Programmer Reference, UP-7621 (see Preface) contains a description of the translation rules.

L.8.2.1. An Example of a User Comparison Subroutine

For the following example, assume the first five characters of each record contains a signed, nonzero number between -49999 and 49999. A negative number X is represented by 50000-X. If key translation is not used, the following comparison subroutine could be used:

```
      SUBROUTINE COMP(FIRST,SECOND,CODE)
      INTEGER FIRST(2),SECOND(2),CODE,F,S
      DECODE(1,FIRST) F
      DECODE(1,SECOND) S
1     FORMAT(15)
      IF (F.GE.50000) F=50000-F
      IF (S.GE.50000) S=50000-S
      IF (F-S) 2,3,4
2     CODE=1
      RETURN
3     CODE=2
      RETURN
4     CODE=3
      RETURN
      END
```

L.8.2.2. An Example of a Runstream with a Comparison Subroutine

In the following example of a comparison subroutine, the first two characters of the records given to the comparison subroutine contain an integer that indicates the starting position of the key within the record. The key is five characters long and contains a right-justified integer value. A complete runstream for using this comparison subroutine is:

```
@RUN
@FTN,SI

      EXTERNAL COMP
      CALL FSORT('comp,rsz=80,core=20000!',9,10,COMP)
      END

@FTN,SI  COMP

      SUBROUTINE COMP(R1,R2,CODE)
      INTEGER CODE
      CHARACTER R1*80,R2*80,F1*8,F2*8
*      compute the key values
      DECODE(4,R1) I1
      DECODE(4,R2) I2
      ENCODE(8,5,F1) I1
      ENCODE(8,5,F2) I2
      DECODE(F1,R1) I1
      DECODE(F2,R2) I2
*      do the comparisons
      IF (I1-I2) 1,2,3
1      CODE=1
      RETURN
2      CODE=2
      RETURN
3      CODE=3
      RETURN
4      FORMAT(12)
5      FORMAT('(',12,'X,15)')
      END

@MAP,SIF
LIB      ASCII*FTNLIB.
@ASG,A   IN*PUT
@ASG,C   OUT*PUT
@USE     9,IN*PUT
@USE     10,OUT*PUT
@XQT
@FIN
```


L.8.3. User Data Reduction Subroutine

The data reduction subroutine will be called by the Sort/Merge package whenever the Sort/Merge package has found two records whose order is immaterial. The data reduction subroutine may or may not merge the two records into the first record. The data reduction subroutine may only be specified when sorting fixed length records. The DATA clause must be specified in the information string in the call to FSORT. The *datred* parameter must be present in the call to FSORT. Two restrictions must be remembered:

1. The records, if merged, must always be merged into the first record (that is, the first argument).
2. The data reduction routine may not change any key fields.

The data reduction subroutine will be called with three arguments. The first two arguments are the two records. The third argument is an integer result assigned by the data reduction subroutine with the following possible values:

- The value 1 indicates that the two records have been merged.
- The value 2 indicates that the two records have not been merged.

The Sort/Merge subroutines translate the key fields according to certain rules. The user must exercise care when using a data reduction subroutine together with keys specified in the information string in the call to FSORT. The translation rules are described in the Sort/Merge Programmer Reference, UP-7621 (see Preface).

L.8.3.1. A Simple Example of a Data Reduction Subroutine

The following data reduction subroutine assumes that any input record that contains a 1 in character position 6 will be chosen over any other record. If both records contain a 1 in character position 6, the first record will be chosen over the second record.

```
      SUBROUTINE DATA(FIRST,SECOND,CODE)
      INTEGER CODE
      CHARACTER FIRST*80,SECOND*80,TEST,ONE/1H1/
      DECODE(1,FIRST) TEST
1     FORMAT(5X,A1)
      IF (TEST.NE.ONE) GO TO 2
      CODE=1
      RETURN
2     DECODE(1,SECOND) TEST
      IF (TEST.EQ.ONE) GO TO 3
      CODE=2
      RETURN
3     FIRST=SECOND
      CODE=1
      END
```

L.8.3.2. An Example of a Runstream with a Data Reduction Subroutine

This example with a data reduction subroutine assumes that two records with equal keys will be merged if character position 11 of at least one of the records is blank. The record with the blank in character position 11 will be retained. If both records have character position 11 blank, the first record will be retained.

This runstream has chosen the decision field outside the key fields to avoid any problems with key field translation.

```
@RUN
@FTN,SI

      EXTERNAL DATA
      CALL FSORT('key=(1/5,6/5/d),rsz=80,data reduction
1      user code,core=20000!',9,10,DATA)
      END

@FTN,SI  DATA

      SUBROUTINE DATA(R1,R2,CODE)
      INTEGER CODE, BLANK/1H /
      CHARACTER*80 R1,R2
      DECODE(1,R1) IB
1      FORMAT(10X,A1)
      IF (IB.NE.BLANK) GO TO 2
      CODE=1
      RETURN
2      DECODE(1,R2) IB
      IF (IB.EQ.BLANK) GO TO 3
      CODE=2
      RETURN
3      R1=R2
      CODE=1
      END

@MAP,SIF
LIB      ASCII*FTNLIB.
@ASG,A   IN*PUT
@ASG,C   OUT*PUT
@USE     9,IN*PUT
@USE     10,OUT*PUT
@XQT
@FIN
```

L.8.4. User-Specified Output Subroutine

The output subroutine will be called by the Sort/Merge package whenever a record is to be written. The output subroutine is called with two arguments. The first argument is the record to be written and the second argument is the length in characters of the record to be written. The second argument is also a flag to the user output subroutine to indicate when the sort has delivered the last record to be written. The length is normally a positive number indicating the size of the record in characters. If the length is negative or zero, the last record has been delivered to the output subroutine.

L.8.4.1. A Simple Example of a User-Specified Output Subroutine

The following output subroutine will output records to unit 6 through a formatted write:

```
      SUBROUTINE OUT(RECORD,LENGTH)
      CHARACTER*80 RECORD
      IF (LENGTH.GT.0) PRINT 1,RECORD
1     FORMAT(1X,A)
      RETURN
      END
```

L.8.4.2. The CALL Statement to FSTAKE

The user output routine may want to indicate to the Sort/Merge package when the output routine needs a new output record. This is done by a CALL statement to FSTAKE with two arguments. The first argument is the record to be received from the Sort/Merge package. The second argument is the length in characters of the new record. If the length argument is negative after returning from FSTAKE, the last record has been delivered from the Sort/Merge package.

L.8.4.3. An Example of FSTAKE in an Output Subroutine

The following example will move records containing one word each into card images with exactly one space between the words and then write the record when the card image becomes full. The sorted records are assumed to contain 80 characters.

```
      SUBROUTINE OUT(RECORD,LENGTH)
      INTEGER POS/1/
      CHARACTER CARD(80),CR(80),BLANK/' '/,RECORD*80
      IF (LENGTH.LT.0) GO TO 99
      * Blank the output record
      DO 1 I=1,80
1     CARD(I)=BLANK
      * Place each character of the record in a word
2     DECODE(80,3,RECORD) CR
3     FORMAT(80A1)
      * Find actual length of record
      DO 4 IL=80,1,-1
4     IF (CR(IL).NE.BLANK) GO TO 5
      * This is a blank record---ignore the record
      GO TO 10
5     IF (POS+IL.LE.81) GO TO 8
      * The card image to print is full--go print it
      PRINT 6,CARD
```

```
6   FORMAT(1X,80A1)
    DO 7 I=1,80
7     CARD(I)=BLANK
    POS=1
8   DO 9 I=1,IL
9     CARD(POS-1+I)=CR(I)
    POS=POS+IL+1
*   get next record to get next word
10  CALL FSTAKE(RECORD,LENGTH)
    IF (LENGTH.GE.0) GO TO 2
99  IF (POS.GT.1) PRINT 6,CARD
    RETURN
    END
```

L.9. Optimizing Sorts

An understanding of this subsection is not required to do a sort. This information is provided for those who need to sort larger data sets than the standard scratch assignments (main storage and mass storage) allow. This information will also help those who need to minimize the resources used in a sort. The user also will need to use this information if the Sort/Merge package error B5 is given for a sort.

The standard scratch file assignments are:

1. 19,000 words of main storage
2. Six disk files of 512 tracks each (initial reserve 0)

This amount of storage should be sufficient to sort some 200,000 to 250,000 card images. If a very large sort (that is, a multiple cycle sort requiring operator intervention) is necessary, the user should consult the Sort/Merge Programmer Reference, UP-7621 (see Preface). The performance of a sort is mainly determined by the following three factors:

1. The bias of the input data
2. The size of the main storage scratch area
3. The scratch files used

The CPU time used by the sort may be decreased slightly by inhibiting the checksum on the sort's scratch files or by increasing the size of the checksum mesh.

L.9.1. The Bias of the Input Data

The bias may be defined as the average number of records in sorted subsequences present in the input file. (See $BIAS = biasno$ in L.4.1.) For example, if the input file is exactly in reverse order, the bias will be 1. Also, random data has a bias of 2. Generally, the bias will be greater if the input file is almost sorted; that is, the more nearly sorted the input file, the greater the bias.

If a bias is specified, the sort will be able to use available resources optimally so that more data can be sorted using the same amount of scratch storage. The user should specify the bias whenever it is known and when the bias is less than 1.4 or greater than 3.

The bias is specified by the form:

$BIAS = biasno$

L.9.2. The Size of the Main Storage Scratch Area

The size of the main storage scratch area may be specified in the following two ways:

1. Assign the file $R\$CORE$ with a suitable maximum granule value.
2. Specify the size of the main storage scratch area in the information string for the sort or merge.

If the size of the main storage scratch area is given by both methods, the $R\$CORE$ value will override the size of the main storage scratch area given in the information string. (See L.4.1.)

L.9.2.1. The Use of $R\$CORE$

The size of the main storage scratch area in words may be specified at run time by assigning the file $R\$CORE$ with a suitable maximum size. For example, if 20,000 words of main storage scratch area is desired, the following control statement will guarantee that 20,000 words of storage are available to the Sort/Merge interface and package:

$@ASG,T R\$CORE,///20$

L.9.2.2. The Use of the CORE Clause in the Information String

The amount of main storage scratch area for the sort is specified by the following CORE clause in the information of the call to FSORT:

$CORE = corsz$

where $corsz$ is the size of the main storage scratch area in words. The Sort/Merge interface rejects any size that is less than 3000 words. The sort will generally execute faster when it is given more main storage.

L.9.3. The Scratch Files Used and Checksum

The use of tape scratch files should be avoided whenever possible. Tape sorts are slower and require operator intervention. The Sort/Merge package distinguishes two cases for mass storage files:

1. One or two mass storage files
2. More than two mass storage files

The first case is more suitable when only one or two mass storage units are available to the sort. However, this case requires a careful assignment of main storage and mass storage scratch resources. The optimal amount of main storage will depend on how much mass storage is available to the sort. A suitable assignment of facilities is given in Appendix E of the Sort/Merge Programmer Reference, UP-7621 (see Preface). Different mass storage scratch files should be kept on separate mass storage units if possible.

L.9.3.1. Scratch Files Named in the Information String

Scratch files may be specified by the FILE clause in the information string. (See L.4.1.) The following restrictions apply when the FILE clause is used:

1. All scratch files must be assigned when the sort is started.
2. A maximum of 26 scratch files may be specified.
3. At least three tape scratch files must be used if any tape scratch files are used.
4. If tape scratch files are used, a maximum of two mass storage files will be used for the sort.

L.9.3.2. Checksum and the Sort

A checksum is normally done on all tape and mass storage files. The user may omit the checksum on one or more device types (mass storage or tape). The user may also specify a checksum mesh size for each device type. For example, if a mesh size of 5 is given, only every fifth word of each block written to tape or mass storage is included in the checksum.

The user may omit the checksum by specifying the nocheck clause (NOCH) in the information string for the call to FSORT. The user may provide a mesh size by specifying the MESH clause in the information string. These clauses are described in the CALL statement to FSORT. (See L.4.1.)

L.10. Sorting Very Large Amounts of Data

When it is not practical to assign enough scratch storage to hold all of the data to be sorted, a multicycle sort must be done. For that case, the ASCII FORTRAN program must be executed with the P option (@XQT,P) and some Sort/Merge package parameter data images must be prepared. These data images are fully described in the Sort/Merge Programmer Reference, UP-7621 (see Preface).

The SMRG parameter data image format is:

```
'SMRG', 'outptprefx', nbrrecds, nbrreel
```

where:

- outptprefx* is a string two characters long that identifies the intermediate output tapes.
- nbrrecds* specifies the number of records to be sorted in each cycle and is optional.
- nbrreel* specifies the number of reels to be produced in each cycle and is optional. If the number given in *nbrrecds* specifies more records than the assigned hardware can hold, *nbrrecds* will be ignored.

The parameter data images are read after the call to FSORT but before the first input record is read or before the user input routine is first called.

The following control image must be used after the last Sort/Merge parameter data image:

```
@EOF A
```

A user wishing to rerun interrupted multicycle sorts must refer to the Sort/Merge Programmer Reference, UP-7621 (see Preface).

L.10.1. An Example of a Large Single Cycle Sort

The following runstream contains a sort which must run as efficiently as possible. The records will be in nearly reverse order (the bias will be about 1.2). A checksum will not be done. About 400,000 records must be sorted, so the standard scratch assignments cannot be used. Ample amounts of main storage and mass storage will be available for the sort.

The first step is to calculate the sort volume. This is the record size times the number of records times a safety factor:

$$20 * 400000 * (1 + .1)$$

which would equal about 9 million words.

For a big sort, six equal size files should be used. This would make each file about 1.5 million words or about 850 tracks.

A suitable amount of main storage scratch area would be about 50K words.

The scratch files must be assigned before the sort may begin. The runstream for the sort might be:

```
@RUN
@FTN,S10

      CALL FSORT('rsz=80,key=(1/5,6/5/d),core=50000,
1      bias=1.2,files=(M1,M2),nocheck=dft,
1      files=(m3,m4,m5,m6)!',9,10)
      END

@MAP,SIF
LIB      ASCII*FTNLIB.
@ASG,A   IN*PUT
@ASG,T   OUT*PUT,T,REELNO
@ASG,T   M1,///850
@ASG,T   M2,///850
@ASG,T   M3,///850
@ASG,T   M4,///850
@ASG,T   M5,///850
@ASG,T   M6,///850
@USE     9,IN*PUT
@USE     10,OUT*PUT
@XQT
@FIN
```

L.10.2. An Example of a Multiple Cycle Sort

The following runstream may be used for a multicycle sort. The information is much the same as the large single cycle sort except that there are about 20 million records to be sorted using the same amount of main storage and mass storage. In addition, four scratch tape files will be used.

```
@RUN
@FTN,S10

      CALL FSORT('rsz=80,key=(1/5,6/5/d),core=50000,
1      bias=1.2,file=(M1,M2,T1,T2,T3,T4),noch=dft,
1      files=(m3,m4,m5,m6)!',9,10)
      END

@MAP,SIF
LIB      ASCII*FTNLIB.
@ASG,A   IN*PUT
@ASG,TV  OUT*PUT,U9V/2,REEL1/REEL2/REEL3
@ASG,T   M1,///1117
@ASG,T   M2,///POS/715
@ASG,T   T1,T
@ASG,T   T2,T
@ASG,T   T3,T
@ASG,T   T4,T
@USE     9,IN*PUT
@USE     10,OUT*PUT
@XQT,P
'SMRG', 'EX'
@EOF A
@FIN
```


L.11. Error Messages from a Sort or a Merge

Two different types of error messages may be produced during a sort or a merge. The first type is written to the console and its form is:

XXXX ERROR CODE Y Z

where XXXX is SORT or MERGE, Y is a letter, and Z is a digit. This message is immediately followed by an "ER ERR\$" exit. This type of message is produced by the Sort/Merge subroutines and is described in the Sort/Merge Programmer Reference, UP-7621 (see Preface).

The second type of message is produced by the Sort/Merge interface with the form:

FTN SORT/MERGE ERROR CODE *NW strg*

where *NW* is a 2-digit error code. The error codes and an explanation for each follows. *Strg* is a 4-character string that provides further information on the error.

- 01 The mnemonic in the information string whose first four characters are given in *strg* is not known to the routine called (for example, SELE is not allowed for merges and UNKNOWN is not allowed for sorts or for merges).
- 02 The routine specified in *strg* has been called with the wrong number of arguments.
- 03 The character position of the most significant character of a key is negative or too large.
- 04 A key length is negative or too large.
- 05 An erroneous key type (such as A for a character key) is specified.
- 06 The sorting sequence is neither A, D, nor a null string.
- 07 The word position of the most significant bit of a bit key is negative or too large.
- 08 The bit position of the most significant bit of a bit key is incorrect (0 or greater than 36).
- 09 The translation table in FSSEQ did not contain the full ISO set. *Strg* contains the octal code for the first character found that cannot be translated.
- 10 The maximum record size (RSZ) given in the information string is negative or greater than 65K.
- 11 No record size (RSZ) is specified in the information string for a sort.
- 12 An impossible link size (0 or greater than the maximum record size) is specified.
- 13 No keys and no user comparison routine are given in the information string.
- 14 An error exists in the collating sequence (FSSEQ). The given string is less than 256 characters.
- 15 The auxiliary main storage area is full. For remedial action, please submit a Software User Report (SUR).
- 16 An overflow occurred in the sort parameter table. For remedial action, please submit a Software User Report (SUR).

- 17 At least one key extends beyond the record.
- 18 A bit key of type A, D, G, L, or P does not start on a 6-bit byte boundary.
- 19 The length in bits of a bit key of type A, D, G, L, or P is not divisible by 6.
- 20 An erroneous return code was given on exit from a user comparison routine.
- 21 An erroneous record length was given on exit from a user input routine.
- 22 The link size was not specified for variable length records and no key specifications were given.
- 23 A forbidden character was found in the information string.
- 24 The output routine/file or the user's comparison routine is not in the argument list.
- 25 The COPY specification was given in the information string, but FSCOPY has not been called (or the most recent call had no arguments).
- 26 For a sort, an input file/routine is not in the argument list. For a merge, either too few (less than two) or too many (more than 26) input files/routines have been given in the argument list.
- 27 The bias was given as less than 1.
- 28 The mnemonic whose first four characters are in *strg* has appeared more than once in the information string. If *strg* is RSZ, VRSZ may have appeared before (and vice versa).
- 29 The size of the main storage scratch area has been given as less than 3000 words or greater than 262,141 words.
- 30 An illegal character was given in the NOCH specification (only D, F, K, and T are accepted to the right of the equals sign) in the information string.
- 31 The user's data reduction routine is not in the argument list.
- 32 A given scratch file is not on mass storage. The most common reason is that the file is not assigned to the run.
- 34 More than 26 scratch files were specified in the information string.
- 35 The first member of a select pair (SELE clause) is not greater than the previous pair's second member.
- 36 The second member of a select pair is less than the first member.
- 37 An erroneous return code was given on exit from a user data reduction routine.
- 38 A facility reject status was generated in the attempt to assign one of the sort scratch files. *Strg* specifies which of the six standard files could not be assigned. The next line gives the FAC REJECT code.
- 39 Some keys overlap. *Strg* gives the number of the major key of the pair that found to overlap (the most major key is number 1, the next number 2, etc.).

- 40 An illegal sign was found in an arithmetic field. *Strg* gives the first four characters found after (and including) the one in error.
- 41 An illegal character was found in a numeric field. *Strg* gives the first four significant digits.
- 42 The field in *strg* was not followed by an equals sign.
- 43 A numeric field given in *strg* was found when an alphabetic field was expected.
- 44 Nonblank characters appeared between an equals sign and a left parenthesis.
- 45 The first field of a *keyspn* in a KEY clause was alphabetic and was not BIT.
- 46 An alphabetic field in *strg* was found when a numeric field was expected.
- 47 An integer field contained a decimal point.
- 48 An invalid delimiter given in *strg* was found.
- 49 The name of a scratch file contained more than 12 characters.
- 50 The first instruction of a user routine is illegal.
- 51 Too many parameters in the call to FSORT or FMERGE exist.
- 52 No main storage scratch parameter argument was given and an OWN clause is in the information string.
- 53 The data reduction routine was specified for a sort of variable length records.
- 55 The first word of the user-provided sort parameter table was wrong.
- 63 The mesh size was previously specified for a device given in *strg* (*strg* will be a value D, F, K, or T).
- 64 An impossible mesh size was specified.
- 65 An illegal device type in *strg* is used in a MESH specification.
- 66 An illegal delimiter is used in a MESH specification.
- 67 Banked data arguments are not allowed.
- 68 Only the first argument to FSORT, FMERGE, FSGIVE, or FSTAKE may be of type CHARACTER.

Index

Term	Reference	Page	Term	Reference	Page
A					
ABORT\$	7.3.3.16.2	7-38	ANSI, OPEN statement	5.10.1	5-57
ABS intrinsic function	Table 7-2	7-9	ARCOS intrinsic function	Table 7-2	7-8
ACOB interface	K.3	K-4	ARGCHK=OFF option	8.5	8-7
ACOS intrinsic function	Table 7-2	7-8	ARGCHK=ON option	8.5	8-7
ACSF\$	7.3.3.16.2	7-38	Argument		
Actual array	2.2.2.4.4	2-12	function	7.2.1	7-2
ADATE	7.3.3.16.2	7-37	subroutine	7.2.2	7-3
	7.3.3.16.2	7-39	type checking	7.5	7-63
Adjustable array	2.2.2.4.3	2-11	K.4.1	K-7	
AIMAG intrinsic function	Table 7-2	7-11	Arithmetic		
AINTE intrinsic function	Table 7-2	7-10	expression	2.2.3.1	2-14
ALGAMA intrinsic function	Table 7-2	7-9	operator	2.2.3.1.1	2-14
Alignment, storage	6.9.1	6-24	primary	2.2.3.1.2	2-15
ALOG intrinsic function	Table 7-2	7-8	term	2.2.3.1.2	2-15
ALOG10 intrinsic function	Table 7-2	7-8	Arithmetic assignment		
AMAX0 intrinsic function	Table 7-2	7-9	statement	3.2	3-1
AMAX1 intrinsic function	Table 7-2	7-9	Array		
AMINO intrinsic function	Table 7-2	7-9	assumed size	2.2.2.4.1	2-11
AMIN1 intrinsic function	Table 7-2	7-9	declaration	2.2.2.4.3	2-11
AMOD intrinsic function	Table 7-2	7-9	dimension	2.2.2.4	2-10
Ampersand			element reference	2.2.2.4.5	2-12
EXTERNAL option	7.2.3	7-3	location of elements	2.2.2.4.6	2-12
for concatenation	2.2.3.2	2-18	subscript	2.2.2.4	2-10
statement label	7.2.1	7-2	ARSIN intrinsic function	Table 7-2	7-8
subprogram name	6.6	6-15	ASCII		
subroutine statement			character set	Appendix B	
label	7.2.2	7-3	FORTRAN	1.1	1-1
AND intrinsic function	7.3.1	7-6	symbiont files	G.4	G-14
ANINT intrinsic function	Table 7-2	7-10	ASCII FORTRAN compiler		
ANSI tape format			calling	10.5	10-21
file processing	G.3.2	G-13	checkout	10.6	10-25
general	5.6.6	5-39	ASIN intrinsic function	Table 7-2	7-8
interchange tapes	G.3.3	G-13	Assembler interface	K.4	K-6
usage	G.3	G-10	Assembly language	1.2	1-2
			ASSIGN statement	4.2.3	4-5

Term	Reference	Page
Assigned GO TO Assignment	4.2.3	4-5
arithmetic	3.2	3-1
character	3.3	3-5
common	6.5	6-13
data storage	6.9.1	6-24
dimension	6.2	6-2
equivalence	6.4	6-10
general	3.1	3-1
initial value	6.8	6-19
logical	3.4	3-6
statement	3.1	3-1
statement label	3.5	3-7
type	6.3	6-4
Associated variables	5.7.1	5-43
Assumed-size array	2.2.2.4.1	2-11
	2.2.2.4.3	2-11
Asterisk	7.4.2.2	7-60
	7.4.3.2	7-62
EXTERNAL option	7.2.3	7-3
fill	5.3.1	5-12
statement label	7.2.1	7-2
subroutine statement label	7.2.2	7-3
AT statement	9.3	9-6
ATAN intrinsic function	Table 7-2	7-8
ATAN2 intrinsic function	Table 7-2	7-8
Automatic storage	8.5.1	8-8
Auxiliary input/output statements	5.10	5-57
CLOSE	5.10.2	5-68
INQUIRE	5.10.3	5-70
OPEN	5.10.1	5-57
Aw format	5.3.1	5-11
B		
BACKSPACE statement	5.6.3	5-36
BANK statement	6.6	6-15
	H.2.3.4.1	H-13
BANKED options	8.5.2	8-9
BANKED=ACTARG option	8.5.2	8-9
BANKED=ALL option	8.5.2	8-9
BANKED=DUMARG option	8.5.2	8-9
BANKED=RETURN option	8.5.2	8-9
Banking	H.2	H-1
BDICALL\$	8.5.3	8-10
BDR	H.2	H-2
BIAS clause	L.4.1	L-6
	L.9.1	L-28
Bit	1.2	1-2
Bit key	L.4.1	L-3
	L.5.1	L-12

Term	Reference	Page
BITS and SBITS	7.3.2.1	7-14
Blank		
format code	5.3.1	5-10
in I/O field	5.3.1	5-10
in numeric input	5.3.10	5-18
INQUIRE	5.10.3	5-73
line	10.4.1.1	10-8
OPEN	5.10.1	5-61
Blank common storage	6.5	6-13
Blank fill	5.3.1	5-9
Blanks	5.3.10	5-18
BLOCK DATA		
procedure	7.8	7-68
program unit	10.2.1	10-1
statement	7.8.2	7-68
structure	7.8.1	7-68
Block IF statement	4.4.1	4-10
execution of	4.4.1.3	4-10
IF-block	4.4.1.2	4-10
IF-level	4.4.1.1	4-10
Block size	5.6.6	5-40
	5.10.1	5-64
	5.10.3	5-75
	G.2.1.3	G-6
Blocking statements		
block IF	4.4.1	4-10
ELSE	4.4.3	4-11
ELSE IF	4.4.2	4-11
END IF	4.4.4	4-12
example	4.4.5	4-12
general	4.4	4-9
BN format	5.3.1	5-10
BOOL intrinsic function	7.3.1	7-6
BREAK checkout command	10.6.2.1	10-26
	10.6.3.1	10-29
Break keyin	10.7.5.2.2	10-64
Buffer offset	5.6.6	5-41
BZ format	5.3.1	5-10
C		
CABS intrinsic function	Table 7-2	7-9
CALL		
checkout command	10.6.2.1	10-27
	10.6.3.2	10-30
control statement	7.2.2	7-3
Calling		
FORTRAN processor	10.5	10-21
subroutine	7.2.2	7-3
Carriage control	5.3.4	5-14
	5.3.10	5-18
CCOS intrinsic function	Table 7-2	7-8
CCOSH intrinsic function	Table 7-2	7-9

Term	Reference	Page	Term	Reference	Page
CDABS intrinsic function	Table 7-2	7-9	general	1.3.1	1-3
CDCOS intrinsic function	Table 7-2	7-8	restrictions	10.6	10-25
CDCOSH intrinsic function	Table 7-2	7-9	soliciting input	10.6.5	10-49
CDEXP intrinsic function	Table 7-2	7-8	Checksum	10.6.2.2	10-27
CDLOG intrinsic function	Table 7-2	7-8	CHKRS\$ service subroutine	L.9.3.2	L-29
CDSIN intrinsic function	Table 7-2	7-8	CHKRS\$ subroutine	7.3.3.11	7-28
CDSINH intrinsic function	Table 7-2	7-8	CHKSV\$ service subroutine	10.6.6	10-49
CDSQRT intrinsic function	Table 7-2	7-8	CHKSV\$ subroutine	7.3.3.11	7-28
CDTAN intrinsic function	Table 7-2	7-8	CHKSV\$ subroutine	10.6.3.12	10-41
CDTANH intrinsic function	Table 7-2	7-9		10.6.6	10-49
CEXP intrinsic function	Table 7-2	7-8	CLEAR checkout command	10.6.3.3	10-32
CHAR intrinsic function	Table 7-2	7-10	CLOG intrinsic function	Table 7-2	7-8
Character			CLOSE service subroutine	7.3.3.18	7-47
assignment statement	3.3	3-5	CLOSE statement	5.10.2	5-68
constant	2.2.1.5	2-5	reread	5.10.2	5-69
conversion	3.3	3-5	CMLPX intrinsic function	Table 7-2	7-10
expression	2.2.3.2	2-17	COBOL interface	K.3	K-4
operator	2.2.3.2	2-17	Code reordering	8.5	8-7
storage	6.9.1	6-24		8.5.4	8-10
substring	2.2.2.5	2-13	Collection		
CHARACTER FUNCTION			and execution	10.5.2.2	10-24
statement	7.4.2.2	7-60	banking	Appendix H	
Character key	L.4.1	L-3	BLOCK DATA	7.8.1	7-68
	L.5.1	L-12	Colon in I/O list	5.3.7.2	5-16
Character set			Comment		
ASCII	Appendix B		convention	1.5	1-6
FORTRAN	2.1	2-1	inline	2.2.7	2-24
CHARACTER type statement	6.3.2.2	6-8	line	2.2.5	2-23
Checkout commands			treatment	10.4.1.1	10-8
BREAK	10.6.3.1	10-29	Common assignment	6.5	6-13
CALL	10.6.3.2	10-30	Common block		
CLEAR	10.6.3.3	10-32	listing	10.4.2.2.6	10-19
DUMP	10.6.3.4	10-33	name	6.5	6-13
EXIT	10.6.3.5	10-34	COMMON statement	6.5	6-13
GO	10.6.3.6	10-35	COMP clause	L.4.1	L-4
HELP	10.6.3.7	10-36		L.5.1	L-14
LINE	10.6.3.8	10-37	Comparison subroutine		
LIST	10.6.3.9	10-37	parameter	L.4.1	L-7
PROG	10.6.3.10	10-38		L.5.1	L-15
RESTORE	10.6.3.11	10-39	Compilation listing		
SAVE	10.6.3.12	10-41	contents	10.4.2.2	10-11
SET	10.6.3.13	10-42	diagnostic messages	10.10	10-70
SETBP	10.6.3.14	10-43	general	10.4.2	10-10
SNAP	10.6.3.15	10-44	options	10.4.2.1	10-10
STEP	10.6.3.16	10-45	with EDIT	8.4	8-6
TRACE	10.6.3.17	10-45	Compilation process		
WALKBACK	10.6.3.18	10-46	general	1.3.1	1-3
Checkout mode	10.6	10-25	with BANK	6.6	6-15
calling	10.6.1	10-25	with checkout	10.6	10-25
contingencies	10.6.4	10-48	with COMPILE	8.5	8-7
debug commands	10.6.3	10-27	with DELETE	8.3	8-5
diagnostics	Appendix I		with INCLUDE	8.2	8-1
entering	10.6.2.1	10-26			

Term	Reference	Page	Term	Reference	Page
Compiler			Control statement		
calling	10.5	10-21	CONTINUE	4.6	4-21
checkout	10.6	10-25	DO	4.5	4-14
location counter usage	6.9.2	6-27	END	4.9	4-24
optimization	10.8	10-67	general	Section 4	
options	10.5.1	10-21	GO TO	4.2	4-2
COMPILER statement	8.5	8-7	IF	4.3	4-7
ARGCHK=OFF	8.5	8-7		4.3.2	4-8
ARGCHK=ON	8.5	8-7	PAUSE	4.7	4-22
BANKED=ACTARG	8.5	8-7	RETURN	7.6	7-65
BANKED=ALL	8.5	8-7	STOP	4.8	4-23
BANKED=DUMARG	8.5	8-7	Conventions of notation	1.5	1-6
BANKED=RETURN	8.5	8-7	Conversion		
DATA=AUTO	8.5	8-7	arithmetic assignment	3.2	3-2
DATA=REUSE	8.5	8-7	ASCII	7.3.3.19	7-47
LINK=IBJ\$	8.5	8-7	character	3.3	3-5
PARMINIT=INLINE	8.5	8-7	E, D, F, and G editing	5.3.10	5-18
PROGRAM=BIG	8.5	8-7	Fieldata	7.3.3.19	7-47
STD=66	8.5	8-7	table	Appendix E	
U1110=OPT	8.5	8-7	COPY clause	L.4.1	L-4
COMPL intrinsic function	7.3.1	7-6		L.5.1	L-14
Complex				L.6.1	L-16
constant	2.2.1.3	2-5	CORE clause	L.4.1	L-5
in format list	5.3.5	5-15	Core parameter	L.4.1	L-7
storage	6.9.1	6-24		L.5.1	L-15
COMPLEX FUNCTION			COS intrinsic function	Table 7-2	7-8
statement	7.4.2.2	7-60	COSH intrinsic function	Table 7-2	7-9
COMPLEX type statement	6.3.2.1	6-7	COTAN intrinsic function	Table 7-2	7-8
Concatenation	2.2.3.2	2-17	Cross reference listing	10.4.2.2.3	10-17
COND\$	7.3.3.16.2	7-37	CSIN intrinsic function	Table 7-2	7-8
	7.3.3.16.2	7-39	CSINH intrinsic function	Table 7-2	7-8
CONJG intrinsic function	Table 7-2	7-11	CSQRT intrinsic function	Table 7-2	7-8
CONS clause	L.4.1	L-2	CTAN intrinsic function	Table 7-2	7-8
Constant	2.2.1	2-3	CTANH intrinsic function	Table 7-2	7-9
character	2.2.1.5	2-5	Currency symbol		
complex	2.2.1.3	2-5	entry name	7.7	7-66
double precision	2.2.1.2.2	2-4	in a symbolic name	2.2.2	2-6
Fieldata	2.2.1.6	2-6	statement label	7.2.1	7-2
	6.8.4	6-20	subroutine statement		
Hollerith	2.2.1.5	2-5	label	7.2.2	7-3
integer	2.2.1.1	2-3	C2F\$	H.2.1.1	H-5
logical	2.2.1.4	2-5			
octal	2.2.1.6	2-6			
real	2.2.1.2	2-3			
single precision	2.2.1.2.1	2-3			
Contingency clause					
checkout	10.6.4	10-48			
general I/O	5.8	5-49			
input/output	5.8.1	5-49			
Continuation line	2.2.6	2-23			
	10.4.1.2	10-8			
CONTINUE statement	4.6	4-21			
			D		
			D (editing code)	5.3.1	5-10
			DABS intrinsic function	Table 7-2	7-9
			DACOS intrinsic function	Table 7-2	7-8
			DARCOS intrinsic function	Table 7-2	7-8
			DARSIN intrinsic function	Table 7-2	7-8
			DASIN intrinsic function	Table 7-2	7-8
			DATA		
			clause	L.4.1	L-4
			options	8.5.1	8-8
			statement	6.8.1	6-19

Term	Reference	Page	Term	Reference	Page
Data			implicit statement	6.3.1	6-4
conversion	Appendix E		implied via names	2.2.2.2.1	2-8
declarations, in block	7.8	7-68	implied via statement	2.2.2.2.2	2-9
initializations	6.8	6-19	of array	2.2.2.4.1	2-10
lengths	6.3	6-4	of dimension	6.2	6-2
reduction subroutine	L.8.3	L-24	DECODE statement	5.9.2	5-52
reduction subroutine			Define file block usage	G.10	G-32
parameter	L.4.1	L-7	DEFINE FILE statement		
storage	6.9.1	6-24	direct	5.7.1	5-43
type statement	6.3	6-4	sequential	5.6.6	5-39
value assignment	6.8.1	6-19	DEFINE statement	7.4.1.1	7-56
DATAN intrinsic function	Table 7-2	7-8	DELETE statement	8.3	8-5
DATAN2 intrinsic function	Table 7-2	7-8	DELL clause	L.4.1	L-3
DATA=AUTO option	8.5.1	8-8	DERF intrinsic function	Table 7-2	7-9
DATA=REUSE option	8.5.1	8-8	DERFC intrinsic function	Table 7-2	7-9
DATE\$	7.3.3.16.2	7-37	DEXP intrinsic function	Table 7-2	7-8
DBANK	H.2	H-1	DFLOAT intrinsic function	Table 7-2	7-10
DBLE intrinsic function	Table 7-2	7-10	DGAMMA intrinsic function	Table 7-2	7-9
DCMPLX intrinsic function	Table 7-2	7-10	Diagnostic		
DCONJG intrinsic function	Table 7-2	7-11	checkout compiler	Appendix I	
DCOS intrinsic function	Table 7-2	7-8	FTNPMD	10.7.5.3	10-65
DCOSH intrinsic function	Table 7-2	7-9	general	10.10	10-70
DCOTAN intrinsic function	Table 7-2	7-8	input/output general	5.8.2	5-50
DDIM intrinsic function	Table 7-2	7-10	messages	Appendix D	
Debug commands	10.6.3	10-27	tables	10.7.2	10-50
Debug facility			Differences between		
AT	9.3	9-6	FORTRAN processors	Appendix A	
DEBUG	9.2	9-2	DIM intrinsic function	Table 7-2	7-10
DISPLAY	9.6	9-9	DIMAG intrinsic function	Table 7-2	7-11
example	9.7	9-10	Dimension		
TRACE OFF	9.5	9-8	adjustable	2.2.2.4.1	2-11
TRACE ON	9.4	9-7	of array	2.2.2.4.1	2-11
Debug mode	10.5.2.1	10-24	value	2.2.2.4.2	2-11
	10.6.2.1	10-26	DIMENSION statement	6.2	6-2
DEBUG statement			DINT intrinsic function	Table 7-2	7-10
general	Section 9		Direct access I/O		
INIT	9.2.4	9-5	DEFINE FILE	5.7.1	5-43
SUBCHK	9.2.2	9-3	FIND	5.7.4	5-48
SUBTRACE	9.2.5	9-5	general	5.7	5-43
TRACE	9.2.3	9-4	READ	5.7.2	5-45
UNIT	9.2.1	9-3	record number	5.2.2	5-4
Debugging			SDF files	G.2.2.2	G-8
checkout	10.6	10-25	WRITE	5.7.3	5-46
	10.6.2	10-26	DISPLAY statement	9.6	9-9
diagnostic system	10.10	10-70	Divide fault	7.3.3.9	7-25
FTNPMD	10.7	10-50	DIVSET service subroutine	7.3.3.9	7-25
FTNWB	10.7	10-50	DLGAMA intrinsic function	Table 7-2	7-9
nonfull	7.3.3.11	7-28	DLOG intrinsic function	Table 7-2	7-8
see truncation problems	9.2	9-2	DLOG10 intrinsic function	Table 7-2	7-8
walkback	10.7	10-50	DMAX1 intrinsic function	Table 7-2	7-9
Declaration			DMIN1 intrinsic function	Table 7-2	7-9
explicit	2.2.2.2.3	2-9	DMOD intrinsic function	Table 7-2	7-9
explicit statement	6.3.2	6-6	DNINT intrinsic function	Table 7-2	7-10

Term	Reference	Page	Term	Reference	Page
DO statement	4.5	4-14	<i>Ew.dDe</i>	5.3.1	5-10
active and inactive	4.5.3	4-15	<i>Ew.dEe</i>	5.3.1	5-10
examples		4-19	<i>Fw.d</i>	5.3.1	5-9
execution	4.5.4	4-16	<i>Gw.d</i>	5.3.1	5-12
extended range	4.5.5	4-18	<i>Gw.dEe</i>	5.3.1	5-12
nested	4.5.2	4-14	' <i>h₁h₂...h_w</i> '	5.3.1	5-11
range of	4.5.1	4-14	<i>lw</i>	5.3.1	5-9
Dollar sign	See		<i>lw.d</i>	5.3.1	5-9
	currency		<i>Jw</i>	5.3.1	5-9
	symbol		<i>Lw</i>	5.3.1	5-10
Double precision			<i>Ow</i>	5.3.1	5-11
complex constants	2.2.1.3	2-5	<i>pP</i>	5.3.1	5-10
real constants	2.2.1.2.2	2-4	<i>Rw</i>	5.3.1	5-11
DOUBLE PRECISION			<i>S</i>	5.3.1	5-10
FUNCTION statement	7.4.2.2	7-60	<i>SP</i>	5.3.1	5-10
DOUBLE PRECISION type			<i>SS</i>	5.3.1	5-10
statement	6.3.2.1	6-7	<i>TLw</i>	5.3.1	5-12
DO-variable availability	4.5.6	4-19	<i>TRw</i>	5.3.1	5-12
DPROD intrinsic function	Table 7-2	7-11	<i>Tw</i>	5.3.1	5-12
DREAL intrinsic function	Table 7-2	7-10	<i>wHh₁...h_w</i>	5.3.1	5-11
DSIGN intrinsic function	Table 7-2	7-10	<i>wX</i>	5.3.1	5-11
DSIN intrinsic function	Table 7-2	7-8	Efficiency	H.2.3	H-12
DSINH intrinsic function	Table 7-2	7-8	Efficient programming	10.9	10-69
DSQRT intrinsic function	Table 7-2	7-8	Element reference	2.2.2.4.5	2-12
DTAN intrinsic function	Table 7-2	7-8	ELSE IF statement	4.4.2	4-11
DTANH intrinsic function	Table 7-2	7-9	ELSE IF-block	4.4.2.1	4-11
Dual-PSR	H.2.1	H-2	execution	4.4.2.2	4-11
Dummy array	2.2.2.4.4	2-12	ELSE statement		
DUMP			ELSE-block	4.4.3.1	4-11
checkout command	10.6.3.4	10-33	general	4.4.3	4-11
PMD command	10.7.5.2.1	10-61	ENCODE statement	5.9.4	5-55
service subroutine	7.3.3.1	7-17	END IF statement	4.4.4	4-12
DVCHK service subroutine	7.3.3.3	7-19	END statement	4.9	4-24
<i>Dw.d</i> format	5.3.1	5-10	ENDFILE statement	5.6.4	5-37
			END= clause specification	5.2.7	5-7
			Entry and exit tracing	9.2.5	9-5
			Entry point listing	10.4.2.2.7	10-20
			ENTRY statement	7.7	7-66
			EQUIVALENCE statement	6.4	6-10
			ERF intrinsic function	Table 7-2	7-9
			ERFC intrinsic function	Table 7-2	7-9
			Error message		
			checkout compiler	Appendix I	
			compiler messages	Appendix D	
			I/O general	5.8	5-49
				5.8.2	5-50
			I/O library	10.7.4.1.1	10-52
			I/O messages	G.9	G-20
				G.9.3	G-21
			math library	10.7.4.1.1	10-52
			user program	10.7.4.1.3	10-55
			ERR\$	7.3.3.16.2	7-38
			ERR= clause specification	5.2.6	5-6
EDIT statement	8.4	8-6			
CODE	8.4	8-6			
PAGE	8.4	8-6			
SOURCE	8.4	8-6			
START	8.4	8-6			
STOP	8.4	8-6			
Editing					
codes	5.3.1	5-9			
repetition of codes	5.3.2	5-14			
repetition of groups	5.3.3	5-14			
variable format	5.3.9	5-18			
Editing codes					
<i>Aw</i>	5.3.1	5-11			
BN	5.3.1	5-10			
BZ	5.3.1	5-10			
<i>Dw.d</i>	5.3.1	5-10			
<i>Ew.d</i>	5.3.1	5-10			

Term	Reference	Page	Term	Reference	Page
ERTRAN	7.3.3.16	7-31	EXIT		
Evaluation of expressions			checkout command	10.6.3.5	10-34
arithmetic	2.2.3.1.3	2-16	PMD command	10.7.5.2.2	10-64
logical	2.2.3.3.3	2-20	service subroutine	7.3.3.15	7-30
typeless	2.2.3.4.2	2-22	EXIT\$	7.3.3.16.2	7-38
Ew.d format	5.3.1	5-10	EXP intrinsic function	Table 7-2	7-8
Ew.dDe format	5.3.1	5-10	Explicit declaration		
Ew.dEe format	5.3.1	5-10	general	2.2.2.2.3	2-9
Executable statement	10.3.1	10-5	statement	6.3.2	6-6
Execution			Exponent		
and collection	10.5.2.2	10-24	overflow	7.3.3.4	7-20
in DO	4.5.4.1	4-16	overflow and underflow	7.3.3.6	7-22
order	10.2.4	10-3	underflow	7.3.3.5	7-21
system	1.3.2	1-4	Expression		
tracing	9.4	9-7	arithmetic	2.2.2.4.5	2-12
using checkout	10.5.2.1	10-24	character	2.2.3.1	2-14
Executive Request	7.3.3.16	7-31	general	2.2.3.2	2-17
ABORT\$	7.3.3.16.2	7-38	logical	2.2.3.3	2-18
ACSF\$	7.3.3.16.2	7-38	typeless	2.2.3.4	2-21
ADATE	7.3.3.16.2	7-39	External function		
COND\$	7.3.3.16.2	7-37	entry	7.7	7-66
DATE\$	7.3.3.16.2	7-37	EXTERNAL statement	7.2.3	7-3
ERR\$	7.3.3.16.2	7-38	general	7.4.2.1	7-60
EXIT\$	7.3.3.16.2	7-38	non-FORTRAN	7.4.2	7-60
FABORT	7.3.3.16.2	7-38	return from	7.10	7-70
FACSF	7.3.3.16.2	7-38	return from	7.6	7-65
FCOND	7.3.3.16.2	7-39	External program unit	10.2.2	10-2
FDATE	7.3.3.16.2	7-39	External reference listing	10.4.2.2.8	10-20
FERR	7.3.3.16.2	7-38	EXTERNAL statement	7.2.3	7-3
FEXIT	7.3.3.16.2	7-38	External subprogram	7.1	7-2
FIO	7.3.3.16.1	7-32			
FIOI	7.3.3.16.1	7-32	F		
FIOW	7.3.3.16.1	7-32	FABORT	7.3.3.16.2	7-38
FIOWI	7.3.3.16.1	7-32	FACSF	7.3.3.16.2	7-38
FIOXI	7.3.3.16.1	7-32	FACSF2	7.3.3.16.2	7-37
FSETC	7.3.3.16.2	7-39	FALSE logical	2.2.1.4	2-5
FTSWAP	7.3.3.16.1	7-33	FASCFD subroutine	7.3.3.19	7-47
FUNLCK	7.3.3.16.1	7-33	FCOND	7.3.3.16.2	7-37
FWANY	7.3.3.16.1	7-33		7.3.3.16.2	7-39
FWST	7.3.3.16.1	7-33	FDATE	7.3.3.16.2	7-37
general	7.3.3.16.1	7-31		7.3.3.16.2	7-39
IOI\$	7.3.3.16.1	7-32	FERR	7.3.3.16.2	7-38
IOWI\$	7.3.3.16.1	7-32	FEXIT	7.3.3.16.2	7-38
IOW\$	7.3.3.16.1	7-32	FFDASC subroutine	7.3.3.19	7-47
IOXI\$	7.3.3.16.1	7-32	Fieldata constant	2.2.1.6	2-6
IO\$	7.3.3.16.1	7-32		6.8.4	6-20
SETC\$	7.3.3.16.2	7-39	File	5.1	5-1
TSWAP\$	7.3.3.16.1	7-33	CLOSE	5.10.2	5-68
UNLCK\$	7.3.3.16.1	7-33	direct	5.7.1	5-43
WAIT\$	7.3.3.16.1	7-33	OPEN	5.10.1	5-57
WANY\$	7.3.3.16.1	7-33			

Term	Reference	Page	Term	Reference	Page
reference number	5.2.1	5-3	slash	5.3.7.1	5-16
reference table	G.6	G-16	variable format	5.3.9	5-18
sequential	5.6.6	5-39	FORMAT statement	5.3	5-8
skeletonized	5.7.1	5-44	Format, editing codes	5.3.1	5-9
FILE clause	L.4.1	L-5	FORTRAN		
	L.9.3.1	L-29	evolution	1.2	1-2
FIND statement	5.7.4	5-48	execution time system	1.3.2	1-4
FIO	7.3.3.16.1	7-31	I/O guide	Appendix G	
FIOI	7.3.3.16.1	7-31	PROC	8.2	8-2
FIOW	7.3.3.16.1	7-31	processor	1.3.1	1-2
FIOWI	7.3.3.16.1	7-31	sample listing	10.4.2.2.2	10-11
FIOXI	7.3.3.16.1	7-31	sample program	1.4	1-4
FLD changed to BITS	A.3	A-4	system	1.3	1-2
FLOAT intrinsic function	Table 7-2	7-10	FORTRAN V	Appendix A	
Floating-point			FORTRAN V interface	K.1	K-1
overflow	7.3.3.8	7-24	FORTRAN-supplied		
underflow	7.3.3.7	7-23	intrinsic procedure	7.3	7-6
FMERGE	L.5	L-11	procedure	7.3	7-6
bit key	L.5.1	L-12	pseudo-function	7.3.2	7-14
Character key	L.5.1	L-12	service subroutine	7.3.3	7-17
COMP clause	L.5.1	L-14	Free core area	G.7	G-18
comparison subroutine			FSCOPY	L.6	L-16
parameter	L.5.1	L-15	key position	L.6.2	L-17
	L.8.2	L-22	link size	L.6.2	L-17
COPY clause	L.5.1	L-14	record size	L.6.2	L-17
core parameter	L.5.1	L-15	sort parameter table	L.6.1	L-16
data reduction subroutine			FSETC	7.3.3.16.2	7-39
parameter	L.8.3	L-24	FSGIVE	L.8.1.2	L-21
INPU clause	L.5.1	L-14	FSORT	L.4	L-1
input subroutine parameter	L.5.1	L-14	BIAS clause	L.4.1	L-6
	L.8.1	L-20		L.9.1	L-28
KEY clause	L.5.1	L-12	bit key	L.4.1	L-3
output subroutine			character key	L.4.1	L-3
parameter	L.5.1	L-15	COMP clause	L.4.1	L-4
	L.8.4	L-26	comparison subroutine		
RSZ clause	L.5.1	L-11	parameter	L.4.1	L-7
VRSZ clause	L.5.1	L-11		L.8.2	L-22
FMT= clause	5.2.4	5-6	CONS clause	L.4.1	L-2
Format specification			COPY clause	L.4.1	L-4
carriage control	5.3.4	5-14	CORE clause	L.4.1	L-5
complex variables	5.3.5	5-15		L.9.2	L-28
control of record	5.3.7	5-15	core parameter	L.4.1	L-7
defined specifications	5.3	5-8	DATA clause	L.4.1	L-4
editing code repetition	5.3.2	5-14	data reduction subroutine		
editing codes	5.3.1	5-9	parameter	L.4.1	L-7
end of output list test	5.3.7.2	5-16		L.8.3	L-24
FMT= clause	5.2.4	5-6	DELL clause	L.4.1	L-3
general	5.2.4	5-5	FILE clause	L.4.1	L-5
list-directed	5.3	5-8		L.9.3.1	L-29
multiple line formats	5.3.7.1	5-16	input subroutine parameter	L.4.1	L-7
output list fulfillment	5.3.7.2	5-16		L.8.1	L-20
relationships to an I/O list	5.3.8	5-17	KEY clause	L.4.1	L-3
scale factor	5.3.6	5-15	large sort	L.10	L-30

Term	Reference	Page	Term	Reference	Page
MESH clause	L.4.1	L-6	statement function	7.4.1	7-56
	L.9.3.2	L-29	structure	7.4.2.1	7-60
NOCH clause	L.4.1	L-6	subprogram	7.4.2	7-60
	L.9.3.2	L-29	typeless	2.2.3.4.1	2-21
output subroutine parameter	L.4.1	L-7	FUNCTION statement	7.4.2.2	7-60
	L.8.4	L-26	FUNLCK	7.3.3.16.1	7-33
RSZ clause	L.4.1	L-1	FWANY	7.3.3.16.1	7-33
R\$CORE	L.9.2.1	L-28	FWST	7.3.3.16.1	7-31
SELE clause	L.4.1	L-5		7.3.3.16.1	7-33
VRSZ clause	L.4.1	L-2	<i>Fw.d</i> format	5.3.1	5-9
FSSEQ	L.7	L-18	F\$EP	10.7.4.3.1	10-58
user-specified collating sequence	L.7	L-18	F\$INFO	10.7.4.3.2	10-58
FSTAKE	L.8.4.2	L-26	F2ACTIV\$	H.2.1.2	H-6
FSTAT	7.3.3.16.1	7-31	F2DYN\$	7.3.3.23	7-54
	7.3.3.16.1	7-33	F2FCA	L.3.3	L-1
FSYMB	7.3.3.16.1	7-31	G		
@FTN	10.5	10-21	GAMMA intrinsic function	Table 7-2	7-9
FTNPMD	10.7	10-50	Global optimization	10.8.2	10-68
	10.7.5	10-61	GO checkout command	10.6.3.6	10-35
diagnostics	10.7.5.3	10-65	GO TO statement		
initiating	10.7.3	10-51	assigned	4.2.3	4-5
soliciting input	10.7.5.1	10-61	computed	4.2.2	4-3
FTNR	7.3.3.11	7-28	general	4.2	4-2
	10.6.2.1	10-26	unconditional	4.2.1	4-2
	10.6.6	10-49	<i>Gw.d</i> format	5.3.1	5-12
purpose	1.3.1	1-3	<i>Gw.dEe</i> format	5.3.1	5-12
FTNWB	10.7	10-50	H		
calling	10.7.4.1.4	10-56	H format	5.3.1	5-11
general	10.7.4	10-52	HELP checkout command	10.6.3.7	10-36
initiating	10.7.3	10-51	HFIX intrinsic function	Table 7-2	7-10
messages	10.7.4.2	10-57	Hierarchy of operators	2.2.3.5	2-22
procedures	10.7.4.3	10-58	Hollerith		
FTN\$PF	8.2	8-4	constant	2.2.1.5	2-5
FTSWAP	7.3.3.16.1	7-31	format	5.3.1	5-11
	7.3.3.16.1	7-33	representation difference	A.4	A-9
Function			' $h_1 h_2 \dots h_w$ ' format	5.3.1	5-11
alternate entry	7.7	7-66	I		
argument	7.5	7-63	IABS intrinsic function	Table 7-2	7-9
external	7.4.2	7-60	IBANK	H.2	H-1
FORTTRAN-supplied	7.3	7-6	IBJ\$	8.5.3	8-10
initial statement	7.4.2.2	7-61	ICHAR intrinsic function	Table 7-2	7-10
internal	7.4.2	7-60	Identification line	10.4.2.2.1	10-11
intrinsic	7.3.1	7-6	IDFIX intrinsic function	Table 7-2	7-10
non-FORTTRAN	7.10	7-70	IDIM intrinsic function	Table 7-2	7-10
programmer-defined	7.4.2.2	7-60	IDINT intrinsic function	Table 7-2	7-10
programmer-defined procedure	7.4	7-56	IDNINT intrinsic function	Table 7-2	7-10
pseudo-function	7.3.2	7-14			
reference	7.2.1	7-2			
return from	7.6	7-65			

Term	Reference	Page	Term	Reference	Page
IF statement			Interactive postmortem dump	10.7.5	10-61
arithmetic	4.3.1	4-7	diagnostics	10.7.5.3	10-65
general	4.3	4-7	soliciting input	10.7.5.1	10-61
logical	4.3.2	4-8	Interlanguage communication	Appendix K	
IFIX intrinsic function	Table 7-2	7-10	Internal file statements		
IMAG intrinsic function	Table 7-2	7-11	DECODE	5.9.2	5-52
Implicit			ENCODE	5.9.4	5-55
name rule	2.2.2.2.2	2-9	general	5.9	5-51
statement	6.3.1	6-4	READ	5.9.1	5-51
Implied declaration	2.2.2.2.1	2-8	WRITE	5.9.3	5-54
Implied DO	5.2.3	5-5	Internal function	7.4.2	7-60
INCLUDE statement	8.2	8-1	Internal subprogram	6.3.1	6-5
INDEX intrinsic function	Table 7-2	7-11		6.6	6-15
Initial value	6.8	6-19		6.6	6-15
Inline function	7.3.1	7-7		7.1	7-2
Inline procedure	7.3	7-6		10.2.2	10-2
INPU clause	L.5.1	L-14	Interrupt	10.6.2.1	10-26
Input			Intrinsic function		
contingency clauses	5.8.1	5-49	ABS	Table 7-2	7-9
direct READ	5.7.2	5-45	ACOS	Table 7-2	7-8
error messages	5.8.2	5-50	AIMAG	Table 7-2	7-11
format	5.3	5-8	AINT	Table 7-2	7-10
list-directed	5.5.1	5-23	ALGAMA	Table 7-2	7-9
namelist	5.4.2	5-20	ALOG	Table 7-2	7-8
sequential READ	5.6.1	5-26	ALOG10	Table 7-2	7-8
Input subroutine	L.8.1	L-20	AMAX0	Table 7-2	7-9
Input subroutine parameter	L.4.1	L-7	AMAX1	Table 7-2	7-9
	L.5.1	L-14	AMINO	Table 7-2	7-9
Input/output control list			AMIN1	Table 7-2	7-9
END= clause	5.2.7	5-7	AMOD	Table 7-2	7-9
ERR= clause	5.2.6	5-6	AND	7.3.1	7-6
file reference number	5.2.1	5-3	ANINT	Table 7-2	7-10
FMT= clause	5.2.4	5-6	ARCOS	Table 7-2	7-8
format specification	5.2.4	5-5	ARSIN	Table 7-2	7-8
input/output status clause	5.2.8	5-7	ASIN	Table 7-2	7-8
namelist specification	5.2.5	5-6	ATAN	Table 7-2	7-8
record number	5.2.2	5-4	ATAN2	Table 7-2	7-8
REC= clause	5.2.2	5-4	BOOL	7.3.1	7-6
UNIT= clause	5.2.1	5-3	CABS	Table 7-2	7-9
Input/output guide	Appendix G		CCOS	Table 7-2	7-8
Input/output list	5.2.3	5-4	CCOSH	Table 7-2	7-9
implied DO	5.2.3	5-5	CDABS	Table 7-2	7-9
Input/output status clause	5.2.8	5-7	CDCOS	Table 7-2	7-8
Input/output status word	5.8.1	5-49	CDCOSH	Table 7-2	7-9
INQUIRE statement	5.10.3	5-70	CDEXP	Table 7-2	7-8
INT intrinsic function	Table 7-2	7-10	CDLOG	Table 7-2	7-8
Integer			CDSIN	Table 7-2	7-8
constant	2.2.1.1	2-3	CDSINH	Table 7-2	7-8
storage	6.9.1	6-24	CDSQRT	Table 7-2	7-8
	Table 6-4	6-25	CDTAN	Table 7-2	7-8
INTEGER FUNCTION statement	7.4.2.2	7-60	CDTANH	Table 7-2	7-9
INTEGER type statement	6.3.2.1	6-7	CEXP	Table 7-2	7-8
			CHAR	Table 7-2	7-10

Term	Reference	Page	Term	Reference	Page
CLOG	Table 7-2	7-8	GAMMA	Table 7-2	7-9
CMPLX	Table 7-2	7-10	general	7.3.1	7-6
COMPL	7.3.1	7-6	HFIX	Table 7-2	7-10
CONJG	Table 7-2	7-11	IABS	Table 7-2	7-9
COS	Table 7-2	7-8	ICHR	Table 7-2	7-10
COSH	Table 7-2	7-9	IDFIX	Table 7-2	7-10
COTAN	Table 7-2	7-8	IDIM	Table 7-2	7-10
CSIN	Table 7-2	7-8	IDINT	Table 7-2	7-10
CSINH	Table 7-2	7-8	IDNINT	Table 7-2	7-10
CSQRT	Table 7-2	7-8	IFIX	Table 7-2	7-10
CTAN	Table 7-2	7-8	IMAG	Table 7-2	7-11
CTANH	Table 7-2	7-9	INDEX	Table 7-2	7-11
DABS	Table 7-2	7-9	INT	Table 7-2	7-10
DACOS	Table 7-2	7-8	ISIGN	Table 7-2	7-10
DARCOS	Table 7-2	7-8	LEN	Table 7-2	7-11
DARSIN	Table 7-2	7-8	LGAMMA	Table 7-2	7-9
DASIN	Table 7-2	7-8	LGE	Table 7-2	7-11
DATAN	Table 7-2	7-8	LGT	Table 7-2	7-11
DATAN2	Table 7-2	7-8	LLE	Table 7-2	7-11
DBLE	Table 7-2	7-10	LLT	Table 7-2	7-11
DCMPLX	Table 7-2	7-10	LOC	7.3.1	7-6
DCONJG	Table 7-2	7-11	LOG	Table 7-2	7-8
DCOS	Table 7-2	7-8	LOG10	Table 7-2	7-8
DCOSH	Table 7-2	7-9	LOWERC	Table 7-2	7-11
DCOTAN	Table 7-2	7-8	MAX	Table 7-2	7-9
DDIM	Table 7-2	7-10	MAX0	Table 7-2	7-9
DERF	Table 7-2	7-9	MAX1	Table 7-2	7-9
DERFC	Table 7-2	7-9	MIN	Table 7-2	7-9
DEXP	Table 7-2	7-8	MIN1	Table 7-2	7-9
DFLOAT	Table 7-2	7-10	MOD	Table 7-2	7-9
DGAMMA	Table 7-2	7-9	NINT	Table 7-2	7-10
DIM	Table 7-2	7-10	OR	7.3.1	7-6
DIMAG	Table 7-2	7-11	REAL	Table 7-2	7-10
DINT	Table 7-2	7-10	SIGN	Table 7-2	7-10
DLGAMA	Table 7-2	7-9	SIN	Table 7-2	7-8
DLOG	Table 7-2	7-8	SINH	Table 7-2	7-8
DLOG10	Table 7-2	7-8	SNGL	Table 7-2	7-10
DMAX1	Table 7-2	7-9	SQRT	Table 7-2	7-8
DMIN1	Table 7-2	7-9	TAN	Table 7-2	7-8
DMOD	Table 7-2	7-9	TANH	Table 7-2	7-9
DNINT	Table 7-2	7-10	TRMLN	Table 7-2	7-11
DPROD	Table 7-2	7-11	UPPERC	Table 7-2	7-11
DREAL	Table 7-2	7-10	XOR	7.3.1	7-6
DSIGN	Table 7-2	7-10	INTRINSIC statement	7.2.4	7-5
DSIN	Table 7-2	7-8	IOC I/O error status function	5.8.1	5-49
DSINH	Table 7-2	7-8	IOI\$	7.3.3.16.1	7-32
DSQRT	Table 7-2	7-8	IOS I/O error status function	5.8.1	5-49
DTAN	Table 7-2	7-8	IOSTAT=	5.2.8	5-7
DTANH	Table 7-2	7-9	IOU I/O error status function	5.8.1	5-49
ERF	Table 7-2	7-9	IOWI\$	7.3.3.16.1	7-32
ERFC	Table 7-2	7-9	IOW\$	7.3.3.16.1	7-32
EXP	Table 7-2	7-8	IOXI\$	7.3.3.16.1	7-32
FLOAT	Table 7-2	7-10	IO\$	7.3.3.16.1	7-32

Term	Reference	Page
ISIGN intrinsic function	Table 7-2	7-10
<i>lw</i> format	5.3.1	5-9
<i>lw.d</i> format	5.3.1	5-9
J		
<i>Jw</i> format	5.3.1	5-9
K		
KEY clause	L.4.1	L-3
	L.5.1	L-12
	L.7	L-18
L		
Label tracing		
enabling	9.2.3	9-4
initiating	9.4	9-7
terminating	9.5	9-8
Labeled common storage	6.5	6-13
Language elements	2.2	2-1
array	2.2.2.4	2-10
constant	2.2.1	2-2
expression	2.2.3	2-14
operators	2.2.3.1.1	2-14
symbolic name	2.2.2	2-6
variable	2.2.2.3	2-9
Large banks	H.2.1.3	H-7
Large programs	Appendix H	
LBJ	H.2	H-2
LCORF\$	7.3.3.22	7-50
LDJ	H.2	H-2
LEN intrinsic function	Table 7-2	7-11
LGAMMA intrinsic function	Table 7-2	7-9
LGE intrinsic function	Table 7-2	7-11
LGT intrinsic function	Table 7-2	7-11
Library procedure	7.3	7-6
LIJ	H.2	H-2
LINE checkout command	10.6.3.8	10-37
LINK=IBJ\$ option	8.5.3	8-10
LIST checkout command	10.6.3.9	10-37
Listing option	10.4.2.1	10-10
List-directed		
general	5.5	5-23
input	5.5.1	5-23
output	5.5.2	5-25
output statements	5.6.2.4	5-34
PRINT	5.6.2.4	5-35
PUNCH	5.6.2.4	5-35
READ	5.6.1.4	5-29
WRITE	5.6.2.4	5-35

Term	Reference	Page
List-directed WRITE	5.6.2.4	5-34
Literal format	5.3.1	5-11
LLE intrinsic function	Table 7-2	7-11
LLT intrinsic function	Table 7-2	7-11
LOC intrinsic function	7.3.1	7-6
Local optimization	10.8.1	10-67
Location counter	6.9.2	6-27
LOG intrinsic function	Table 7-2	7-8
Logical		
constant	2.2.1.4	2-5
evaluation	2.2.3.3.3	2-20
expression	2.2.3.3	2-18
expression formation	2.2.3.3.2	2-20
factor	2.2.3.3.2	2-20
operator	2.2.3.3.1	2-18
primary	2.2.3.3.2	2-20
storage	Table 6-4	6-25
term	2.2.3.3.2	2-20
Logical assignment statement	3.4	3-6
LOGICAL FUNCTION statement	7.4.2.2	7-60
LOGICAL type statement	6.3.2.1	6-7
LOG10 intrinsic function	Table 7-2	7-8
LOWERC intrinsic function	Table 7-2	7-11
<i>Lw</i> format	5.3.1	5-10

M

Machine language	1.2	1-2
	1.3	1-2
Main program		
BANKED=ACTARG option	8.5.2	8-9
banking	Appendix H	
definition	7.1	7-1
general	10.2.2	10-2
sample	1.4	1-4
MASM interface	K.4	K-6
MAX intrinsic function	Table 7-2	7-9
MAXAD\$ service subroutine	7.3.3.20	7-49
MAX0 intrinsic function	Table 7-2	7-9
MAX1 intrinsic function	Table 7-2	7-9
MCORE\$	H.2.1.1	H-5
MCORF\$	7.3.3.22	7-50
MESH clause	L.4.1	L-6
	L.9.3.2	L-29
Method of storage assignment	6.9	6-24
MIN intrinsic function	Table 7-2	7-9
MIN1 intrinsic function	Table 7-2	7-9
MOD intrinsic function	Table 7-2	7-9
Multibanking	Appendix H	
BANK	6.6	6-15
FORTRAN compiler	1.3.1	1-2

Term	Reference	Page	Term	Reference	Page
N					
Name rule	2.2.2.2.1	2-8	Option		
Namelist			format	5.2.4	5-6
input	5.4.2	5-20	on PDP call	8.2	8-3
name specification	5.2.5	5-6	on processor call	10.5	10-21
output	5.4.3	5-22	OR intrinsic function	7.3.1	7-6
READ	5.6.1.3	5-29	Order of statements	10.3.2	10-6
WRITE	5.6.2.3	5-34	Output		
NAMELIST statement	5.4.1	5-19	contingency clauses	5.8.1	5-49
Nested DO-loop	4.5.2	4-14	direct WRITE	5.7.3	5-46
NINT intrinsic function	Table 7-2	7-10	error messages	5.8.2	5-50
NOCH clause	L.4.1	L-6	format	5.2.4	5-6
	L.9.3.2	L-29	list-directed	5.5.2	5-25
Nonexecutable statement	10.3.1	10-5	namelist	5.4.3	5-22
Non-FORTRAN argument	7.10	7-70	sequential WRITE	5.6.2	5-32
NTRAN\$ service subroutine	7.3.3.17	7-39	Output subroutine parameter	L.4.1	L-7
				L.5.1	L-15
			OVERFL service subroutine	7.3.3.4	7-20
			OVFSET service subroutine	7.3.3.8	7-24
			OVUNFL service subroutine	7.3.3.6	7-22
			Ow format	5.3.1	5-11
			P		
O option	8.5.7	8-12	P format	5.3.1	5-10
Object code listing	10.4.2.2.4	10-18	Paged data banks	H.2	H-1
Object program	1.3	1-2	PARAMETER		
	10.5.2	10-24	in program	10.3.2	10-6
Octal constant	2.2.1.6	2-6	statement	6.7	6-17
	6.8.3	6-20	with DELETE	8.3	8-5
OPEN statement	5.10.1	5-57	PARMINIT=INLINE option	8.5.1	8-8
block size	5.10.1	5-64	PAUSE routine	10.6.2.1	10-26
implicit CLOSE	5.10.1	5-66	PAUSE statement	4.7	4-22
record format	5.10.1	5-61	PCIOS	G.1	G-1
record size	5.10.1	5-63	PDP procedures (entry)	8.2	8-1
reread	5.10.1	5-65	PDUMP service subroutine	7.3.3.2	7-18
segment size	5.10.1	5-65	PL/I interface	K.2	K-2
Operator			PMD	10.7.3	10-51
arithmetic	2.2.3.1.1	2-14		10.7.5.1	10-61
character	2.2.3.2	2-17	PMD mode commands		
concatenation	2.2.3.2	2-17	DUMP	10.7.5.2.1	10-61
hierarchy	2.2.3.5	2-22	EXIT	10.7.5.2.2	10-64
logical	2.2.3.3.1	2-18	PP format	5.3.1	5-10
relational	2.2.3.3.1	2-18	PRINT		
Optimization	H.2.3.4.2	H-14	formatted	5.6.2.1	5-32
code reordering	8.5.4	8-10	list-directed	5.6.2.4	5-35
compiler	10.8	10-67	namelist	5.6.2.3	5-34
general	1.3.1	1-3	PROC	8.2	8-1
global	10.8.2	10-68	Procedure		
local	10.8.1	10-67	FORTRAN-supplied	7.3	7-6
pitfalls	3.2	3-2	general	7.1	7-1
	4.5.6	4-19		10.2.2	10-2
	10.8.3	10-68			

Term	Reference	Page	Term	Reference	Page
non-FORTRAN	7.10	7-70			
references	7.2	7-2			
Procedure Definition					
Processor (PDP)					
general	8.2	8-1			
sample reference	8.2	8-3			
Procedure subprogram	10.2.2	10-1			
Processor, FORTRAN	1.3.1	1-2			
PROG checkout command	10.6.3.10	10-38			
Program					
execution	9.2	9-2			
	10.2.4	10-3			
format	10.4.1	10-7			
main program	7.1	7-1			
organization	10.2	10-1			
subprogram	7.1	7-1			
unit	10.2.1	10-1			
unit organization	10.2.3	10-3			
Program control statement					
COMPILER statement	8.5	8-7			
DELETE	8.3	8-5			
EDIT	8.4	8-6			
general	Section 8				
INCLUDE	8.2	8-1			
PARAMETER	6.7	6-17			
PROGRAM statement	7.9	7-69			
Program unit	2.2.2.1	2-7			
function	7.4.2	7-60			
organization	10.2.3	10-3			
procedures	7.1	7-1			
subroutine	7.4.3	7-61			
types	10.2.2	10-1			
Programmer check list	Appendix C				
Programmer-defined					
procedure					
BLOCK DATA	7.8	7-68			
function	7.4.2	7-60			
statement function	7.4.1	7-56			
subroutine	7.4.3	7-61			
Programming techniques	C.3	C-3			
PROGRAM=BIG option	8.5	8-7			
Pseudo-function					
BITS	7.3.2.1.1	7-14			
general	7.3.2	7-14			
SUBSTR	7.3.2.2	7-16			
PSR window	H.2	H-2			
PSRM	H.2	H-2			
PSRU	H.2	H-2			
PUNCH					
formatted	5.6.2.1	5-32			
list-directed	5.6.2.4	5-35			
namelist	5.6.2.3	5-34			
			R		
			Range of DO-loop	4.5.1	4-14
			READ		
			direct access	5.7.2	5-45
			formatted	5.6.1.1	5-26
			internal file	5.9.1	5-51
			list-directed	5.6.1.4	5-29
			namelist	5.6.1.3	5-29
			reread	5.6.1.5	5-31
			unformatted	5.6.1.2	5-28
			Real		
			constant	2.2.1.2	2-3
			storage	6.9.1	6-24
			REAL FUNCTION statement	7.4.2.2	7-60
			REAL intrinsic function	Table 7-2	7-10
			REAL type statement	6.3.2.1	6-7
			Record		
			buffer offset	5.6.6	5-41
			direct DEFINE FILE	5.7.1	5-43
			end-of-file	5.6.4	5-37
			file	5.6.6	5-39
			form	5.6.6	5-41
			formatted	5.6.1.1	5-27
				5.6.2.1	5-33
			list-directed	5.6.1.4	5-30
				5.6.2.4	5-34
			namelist	5.6.1.3	5-29
				5.6.2.3	5-34
			segment	G.2.1	G-2
			sequential DEFINE FILE	5.6.6	5-39
			size	5.6.6	5-39
				5.10.1	5-63
			unformatted	5.6.1.2	5-28
				5.6.2.2	5-33
			Record number specification	5.2.2	5-4
			REC= clause	5.2.2	5-4
			Reference		
			procedure	7.2	7-2
			subroutine	7.2.2	7-3
			Register usage	K.4.2	K-8
			Relocatable binary output	1.3.1	1-3
			Reread statement	5.6.1.5	5-31
			Restart processor (FTNR)		
			general	10.6.6	10-49
			purpose	1.3.1	1-3
			Restarting program	10.6.3.11	10-39
			RESTORE checkout command	10.6.3.11	10-39
			RETURN statement	7.6	7-65
			REWIND statement	5.6.5	5-38
			RSZ clause	L.4.1	L-1
				L.5.1	L-11

Term	Reference	Page	Term	Reference	Page
Run condition switch	7.3.3.12	7-28	DVCHK	7.3.3.3	7-19
Run condition word	7.3.3.12	7-28	ERTRAN	7.3.3.16	7-31
RUN Executive command	1.4.3	1-6	EXIT	7.3.3.15	7-30
Rw format	5.3.1	5-11	FASCFD	7.3.3.19	7-47
R\$CORE	L.3.3	L-1	FFDASC	7.3.3.19	7-47
	L.9.2	L-28	F2DYN\$	7.3.3.23	7-54
R\$FILE	L.6.1	L-16	LCORF\$	7.3.3.22	7-50
			MAXAD\$	7.3.3.20	7-49
S			MCORF\$	7.3.3.22	7-50
S format	5.3.1	5-10	NTRAN\$	7.3.3.17	7-39
Sample listing			OVERFL	7.3.3.4	7-20
common block	10.4.2.2.6	10-19	OVFSET	7.3.3.8	7-24
entry point	10.4.2.2.7	10-20	OVUNFL	7.3.3.6	7-22
external reference	10.4.2.2.8	10-20	PDUMP	7.3.3.2	7-18
termination message	10.4.2.2.9	10-20	SLITE	7.3.3.13	7-29
Sample program	Figure 1-2	1-5	SLITET	7.3.3.14	7-30
SAVE checkout command	10.6.3.12	10-41	SSWTCH	7.3.3.12	7-28
SAVE statement	7.12	7-72	UNDRFL	7.3.3.5	7-21
Scalars	2.2.1	2-3	UNDSET	7.3.3.7	7-23
	2.2.2.4	2-10	SET checkout command	10.6.3.13	10-42
Scale factor	5.3.6	5-15	SETBP checkout command	10.6.2.1	10-26
Scope of names	7.11	7-70		10.6.3.14	10-43
SDF			SETC\$	7.3.3.16.2	7-39
block size	G.2.1.3	G-6	Side effects of assignments	3.2	3-2
data records	G.2.1.2	G-5	SIGN intrinsic function	Table 7-2	7-10
direct access	G.2.2.2	G-8	Sign-on line	10.6.6	10-49
end-of-file record	G.2.1.4	G-6	SIN intrinsic function	Table 7-2	7-8
file layout	G.2.1.5	G-7	Single precision		
file processing	G.2.2	G-7	complex constants	2.2.1.3	2-5
general	G.2	G-2	real constants	2.2.1.2.1	2-3
labels	G.2.1.1	G-3	Single PSR	H.2.2	H-10
record segments	G.2.1.2	G-5	SINH intrinsic function	Table 7-2	7-8
sequential access	G.2.2.1	G-7	Skeletonized file	5.7.1	5-44
Segment size	5.6.6	5-40	Slash (/)		
	5.10.1	5-65	end-of-record	5.5	5-23
	5.10.3	5-75	in DATA statement	6.8.1	6-19
SELE clause	L.4.1	L-5	in DIMENSION	6.2	6-2
Sequential access I/O			in I/O list	5.3.7.1	5-16
BACKSPACE	5.6.3	5-36	in NAMELIST statement	5.4.1	5-20
DEFINE FILE	5.6.6	5-39	in type statement	6.3.2.1	6-7
ENDFILE	5.6.4	5-37		6.3.2.2	6-8
general	5.6	5-26	SLITE service subroutine	7.3.3.13	7-29
output	5.6.2	5-32	SLITET service subroutine	7.3.3.14	7-30
READ	5.6.1	5-26	SNAP checkout command	10.6.3.15	10-44
REWIND	5.6.5	5-38	SNGL intrinsic function	Table 7-2	7-10
SDF files	G.2.2.1	G-7	Sort parameter table	L.6.1	L-16
Service subroutine			Sort/Merge interface		
CHKRS\$	7.3.3.11	7-28	banked arguments	L.3	L-1
CHKSV\$	7.3.3.11	7-28	checksum	L.9.3	L-29
CLOSE	7.3.3.18	7-47	data reduction subroutine	L.8.3	L-24
DIVSET	7.3.3.9	7-25	error messages	L.11	L-32
DUMP	7.3.3.1	7-17	FMERGE	L.5	L-11
			FSCOPY	L.6	L-16

Term	Reference	Page	Term	Reference	Page
FSGIVE	L.8.1.2	L-21	STD=66 option	8.5.5	8-11
FSORT	L.4	L-1	STEP checkout command	10.6.2.1	10-26
FSSEQ	L.7	L-18		10.6.3.16	10-45
FSTAKE	L.8.4.2	L-26	STOP statement	4.8	4-23
large sort	L.10	L-30	Storage		
optimization	L.9	L-27	alignment	6.4	6-11
output subroutine	L.8.4	L-26	EQUIVALENCE	6.4	6-10
scratch files	L.9.3	L-29	in banks	6.6	6-15
user comparison routine	L.8.2	L-22	of data	6.9.1	6-24
user-specified input			Storage area between units	6.5	6-13
subroutine	L.8.1	L-20	Storage assignment		
user-specified subroutines	L.8	L-20	map	10.4.2.2.5	10-19
Source program	1.3.1	1-2	method	6.9	6-24
definition	1.3	1-2	Storage control table	G.8	G-19
format	10.4.1	10-7		K.4.3	K-8
listing	10.4.2.2.2	10-11	Storage-allocation packet	G.11	G-35
SP format	5.3.1	5-10	Subprogram	10.2.2	10-2
Space fill	5.3.1	5-9	banking	Appendix H	
Specification statement			BLOCK DATA	7.1	7-2
COMMON	6.5	6-13		7.8	7-68
DATA	6.8.1	6-19	definition	7.1	7-1
DIMENSION	6.2	6-2	external	7.1	7-2
EQUIVALENCE	6.4	6-10	function	7.1	7-1
explicit typing	6.3.2	6-6	internal	7.1	7-2
implicit typing	6.3.1	6-5	program unit	10.2.1	10-1
Specification subprogram			subroutine	7.1	7-1
BLOCK DATA	7.8	7-68	Subroutine		
organization	10.2.2	10-2	argument	7.5	7-63
	10.2.3	10-3	BANKED=DUMARG option	8.5.2	8-9
Specification system, data			BANKED=RETURN option	8.5.2	8-9
storage	6.9.1	6-24	CALL	7.2.2	7-3
Specification typing, implicit	6.3.1	6-4	general	7.4.3	7-61
SQRT intrinsic function	Table 7-2	7-8	return from	7.6	7-65
SS format	5.3.1	5-10	service	7.3.3	7-17
SSWTCH service subroutine	7.3.3.12	7-28	structure	7.4.3.1	7-61
Statement			subprogram	10.2.2	10-1
arithmetic IF	4.3.1	4-7	SUBROUTINE statement	7.4.3.2	7-62
categories	10.3	10-5	Subscript checking	9.2.2	9-3
classification	10.3.1	10-5	SUBSTR	7.3.2.2	7-16
composition	10.4.1.2	10-8	Substring expressions	2.2.2.5	2-13
executable	10.3.1	10-5		6.4	6-10
form	2.2.4	2-23	Substring, character	2.2.2.5	2-13
general	10.4.1.2	10-8	Symbolic name		
label	10.4.1.3	10-9	data types	2.2.2.2	2-8
nonexecutable	10.3.1	10-5	format	2.2.2	2-6
ordering	10.3.2	10-6	uniqueness	2.2.2.1	2-7
tables	Appendix F		variable	2.2.2.3	2-9
Statement function			Symbols		
DEFINE	7.4.1.1	7-56	\$	5.4.2	5-20
general	7.4.1	7-56	entry name	7.7	7-66
reference	7.4.1.2	7-58	in a symbolic name	2.2.2	2-6
Statement label	10.4.1.3	10-9	statement label	7.2.1	7-2
assigning value	3.5	3-7	subroutine statement		
general	10.4.1.3	10-9	label	7.2.2	7-3

Term	Reference	Page	Term	Reference	Page
&	5.4.2	5-20	Unit reference number	G.5	G-15
EXTERNAL option for concatenation	7.2.3 2.2.3.2	7-3 2-18	UNIT= clause	5.2.1	5-3
statement label	7.2.1	7-2	UNLCK\$	7.3.3.16.1	7-33
subprogram name	6.6	6-15	UPPERC intrinsic function	Table 7-2	7-11
subroutine statement label	7.2.2	7-3	User comparison routine	L.8.2	L-22
*	7.4.2.2 7.4.3.2	7-60 7-62	User-specified collating sequence	L.7	L-18
EXTERNAL option	7.2.3	7-3	User-specified output subroutine	L.8.4	L-26
statement label	7.2.1	7-2	User-specified subroutines	L.8	L-20
subroutine statement label	7.2.2	7-3	U1110=OPT option	8.5.4	8-10
System Data Format (SDF)			V		
general	5.6.6	5-39	Value change tracing	9.2.4	9-5
OPEN	5.10.1	5-57	Variable	2.2.2.3	2-9
T			Variable format	5.3.9	5-18
TAN intrinsic function	Table 7-2	7-8	Volatile register set	K.4.2	K-8
TANH intrinsic function	Table 7-2	7-9	VRSZ clause	L.4.1	L-2
Termination message	10.4.2.2.9	10-20	L.5.1	L-11	
TLw format	5.3.1	5-12	W		
TRACE checkout command	10.6.3.17	10-45	WAIT\$	7.3.3.16.1	7-33
TRACE OFF	9.5	9-8	Walkback	10.7	10-50
TRACE ON	9.4	9-7	messages	10.7.4	10-52
TRMLEN intrinsic function	Table 7-2	7-11	procedures	10.7.4.2	10-57
TRUE logical	2.2.1.4	2-5	10.7.4.3	10-58	
Truncation errors	H.1	H-1	WALKBACK checkout command	10.6.3.18	10-46
Truncation problems			messages	10.6.3.18	10-47
collection and execution	10.5.2.2	10-24	WANY\$	7.3.3.16.1	7-33
DIMENSION statement	6.2	6-2	wHh ₁ ...h _w format	5.3.1	5-11
initial value assignment	6.8	6-19	Word	1.2	1-2
storage assignment	6.9	6-24	WRITE		
TRw format	5.3.1	5-12	direct access	5.7.3	5-46
TSWAP\$	7.3.3.16.1	7-33	formatted	5.6.2.1	5-32
Tw format	5.3.1	5-12	list-directed	5.6.2.4	5-35
Type rules, arithmetic	2.2.3.1.4	2-16	namelist	5.6.2.3	5-34
Type statement			unformatted	5.6.2.2	5-33
explicit	6.3.2	6-6	WX format	5.3.1	5-11
general	6.3	6-4	X		
implicit	6.3.1	6-4	X format	5.3.1	5-11
Typeless			XOR intrinsic function	7.3.1	7-6
evaluation	2.2.3.4.2	2-22	Z		
expression	2.2.3.4	2-21	Zero-fill	5.3.1	5-9
function	2.2.3.4.1	2-21			
U					
Unary operator	2.2.3.1.1	2-14			
UNDRFL service subroutine	7.3.3.5	7-21			
UNDSET service subroutine	7.3.3.7	7-23			

Term	Reference	Page
\$	5.4.2	5-20
entry name	7.7	7-66
in a symbolic name	2.2.2	2-6
statement label	7.2.1	7-2
subroutine statement label	7.2.2	7-3
&	5.4.2	5-20
EXTERNAL option	7.2.3	7-3
for concatenation	2.2.3.2	2-18
statement label	7.2.1	7-2
subprogram name	6.6	6-15
subroutine statement label	7.2.2	7-3
*	7.4.2.2	7-60
	7.4.3.2	7-62
EXTERNAL option	7.2.3	7-3
statement label	7.2.1	7-2
subroutine statement label	7.2.2	7-3

USER COMMENT SHEET

Comments concerning the content, style, and usefulness of this manual may be made in the space provided below. Please fill in the requested information.

This User Comment Sheet will not normally lead to a reply to the originator. Requests for copies of manuals, lists of manuals, pricing information, etc. must be made through your Series 1100 site manager, to your Sperry Univac representative, or to the Sperry Univac office serving your locality. Software problems should be submitted on a Software User Report (SUR) form UD1-745. Questions of a technical nature regarding either the manual or the software should be submitted on a Technical Question (question/answer) form UD1-1195. These forms are available through your Sperry Univac representative.

Customer Name: _____ System Type: _____

Title of Manual: _____

UP No.: _____ Revision No.: _____ Update: _____

Name of User: _____ Date: _____

Address of User: _____

Comments: Give page and paragraph reference where appropriate.

Please rate this manual.	<u>Good</u>	<u>Adequate</u>	<u>Not Adequate</u>
Organization of the text	_____	_____	_____
Clarity of the text	_____	_____	_____
Adequacy of coverage	_____	_____	_____
Examples	_____	_____	_____
Cross references	_____	_____	_____
Tables	_____	_____	_____
Illustrations	_____	_____	_____
Index	_____	_____	_____
Appearance	_____	_____	_____

YOUR COMMENTS, PLEASE - - - -

This manual is part of a library that serves as a source of information for personnel using SPERRY UNIVAC® systems. Space is provided on the opposite side of this form for your comments concerning the usefulness of the information presented. Each comment will be carefully reviewed by the persons responsible for writing and publishing this manual. All comments and suggestions become the property of Sperry Univac.

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

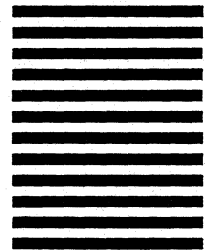
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY
SPERRY UNIVAC

**ATTN: Systems Support
1100 Systems Publications
M.S. 4533**

P.O. Box 43942
St. Paul, Minnesota 55164



CUT

FOLD