Gould FORTRAN 77+

Release 4.1

Reference Manual

July 1984

Publication Order Number 323-004010-000

GOULD
Electronics

ITEM . ACTION ITEM . ACTION ITEM . ACTION ITEM . ACTION ITEM . ACTION ITEM

Gould FORTRAN 77+
Release 4.2
Reference Manual
July 1985

Please remove/insert the appropriate pages as listed below to update Publication Order Number 323-004010-000.

Remove	Insert
Title page - Release 4.1	Title page - Release 4.2
	History page Change 1 in front of iii
v	v Change 1
vii	vii Change 1
ix and x	ix Change 1 and x Change 1
2-1	2-1 Change 1
2-8 through 2-10	2-8 Change 1 through 2-10 Change 1
3-11	3-11 Change 1
7-9	7-9 Change 1
7-14	7-14 Change 1
9-5 through 9-10	9-5 Change 1 through 9-10 Change 1
9-13 through 9-15	9-13 Change 1 through 9-15 Change 1
11-9 and 11-10	11-9 Change 1 and 11-10 Change 1
11-14 and 11-15	11-14 Change 1 and 11-15 Change 1
11-21 through 11-23	11-21 Change 1 through 11-23 Change 1

Continued

CHANGE PACKAGE 1 for 323-004010-000

Change Package Publication Order Number: 323-004010-001

Remove	Insert
11-30	11-30 Change 1
11-35	11-35 Change 1
11-45	11-45 Change 1
11-54	11-54 Change 1
12-6	12-6 Change 1
12-20	12-20 Change 1
13-1 and 13-2	13-1 Change 1 and 13-2 Change 1
13-5/13-6	13-5/13-6 Change 1
14-7/14-8	14-7/14-8 Change 1
15-3	15-3 Change 1
A-2 through A-4	A-2 Change 1 through A-4 Change 1
D-4 through D-8	D-4 Change 1, D-4A/D-4B Change 1 and D-5 Change 1 through D-8 Change 1
D-11	D-11 Change 1
D-15 and D-16	D-15 Change 1 and D-16 Change 1
	E-1 Change 1 through E-6 Change 1
IN-3 through IN-9	IN-3 Change 1 through IN-9 Change 1

Gould FORTRAN 77+

Release 4.2

Reference Manual

July 1985

Publication Order Number: 323-004010-000

GOULD
Electronics

This manual is supplied without representation or warranty of any kind. Gould Inc., Computer Systems Division therefore assumes no responsibility and shall have no liability of any kind arising from the supply or use of this publication or any material contained herein.

LIMITED RIGHTS LEGEND

for

PROPRIETARY INFORMATION

The information contained herein is proprietary to Gould CSD and/or its vendors, and its use, disclosure or duplication is subject to the restrictions stated in the Gould CSD license agreement Form No. 620-06(1/82) or the appropriate third-party sublicense agreement. The information is provided to government customers with limited rights as described in DAR 7-104.9A.

MPX-32 and CONCEPT/32 are Trademarks of Gould, Inc.

Gould FORTRAN 77+

Release 4.1

Reference Manual

July 1984

Publication Order Number 323-004010-000

GOULD
Electronics

This manual is supplied without representation or warranty of any kind. Gould Inc., Computer Systems Division therefore assumes no responsibility and shall have no liability of any kind arising from the supply or use of this publication or any material contained herein.

LIMITED RIGHTS LEGEND

for

PROPRIETARY INFORMATION

The information contained herein is proprietary to Gould CSD and/or its vendors, and its use, disclosure or duplication is subject to the restrictions stated in the Gould CSD license agreement Form No. 620-06(1/82) or the appropriate third-party sublicense agreement. The information is provided to government customers with limited rights as described in DAR 7-104.9A.

HISTORY

The FORTRAN 77+ Release 4.1 Reference Manual, Publication Order Number 323-004010-000, was printed in July 1984.

This manual contains the following pages:

Title page
Copyright page
iii through xi/xii
1-1 through 1-6
2-1 through 2-12
3-1 through 3-18
4-1 through 4-18
5-1 through 5-12
6-1 through 6-29/6-30
7-1 through 7-27/7-28
8-1 through 8-6
9-1 through 9-32
10-1 through 10-2
11-1 through 11-62
12-1 through 12-27/12-28
13-1 through 13-5/13-6
14-1 through 14-7/14-8
15-1 through 15-20
A-1 through A-4
B-1 through B-23/B-24
C-1 through C-7/C-8
D-1 through D-18
IN-1 through IN-10

HISTORY

The FORTRAN 77+ Release 4.1 Reference Manual, Publication Order Number **323-004010-000** , was printed July 1984.

Publication Order Number **323-004010-001** (Change 1, Release 4.2) was printed July 1985.

The updated manual contains the following pages:

	* Change Number
Title Page	1
Copyright Page	0
History Page	1
iii through iv	0
v	1
vi	0
vii	1
viii	0
ix and x	1
xi/xii	0
1-1 through 1-6	0
2-1	1
2-2 through 2-7	0
2-8 through 2-10	1
2-11 and 2-12	0
3-1 through 3-10	0
3-11	1
3-12 through 3-18	0
4-1 through 4-18	0
5-1 through 5-12	0
6-1 through 6-29/6-30	0
7-1 through 7-8	0
7-9	1
7-10 through 7-13	0
7-14	1
7-15 through 7-27/7-28	0
8-1 through 8-6	0
9-1 through 9-4	0
9-5 through 9-7	1
9-8	0
9-9 and 9-10	1
9-11 and 9-12	0
9-13	1
9-14	0
9-15	1
9-16 through 9-32	0
10-1 and 10-2	0
11-1 through 11-8	0
11-9 and 11-10	1
11-11 through 11-13	0
11-14 and 11-15	1
11-16 through 11-20	0
11-21 through 11-23	1
11-24 through 11-29	0
11-30	1
11-31 through 11-34	0
11-35	1
11-36 through 11-44	0
11-45	1
11-46 through 11-53	0
11-54	1
11-55 through 11-62	0
12-1 through 12-5	0
12-6	1
12-7 through 12-19	0
12-20	1
12-21 through 12-27/12-28	0
13-1 and 13-2	1
13-3 through 13-4	0
13-5/13-6	1
14-1 through 14-6	0
14-7/14-8	1
15-1 and 15-2	0
15-3	1
15-4 through 15-20	0
A-1	0
A-2 through A-4	1
B-1 through B-23/B-24	0
C-1 through C-7/C-8	0
D-1 through D-3	0

* Zero in this column indicates an original page.

	* Change Number
D-4 through D-5	1
D-6 and D-7	0
D-8	1
D-9 through D-10	0
D-11	1
D-12 through D-14	0
D-15 and D-16	1
D-17 and D-18	0

	* Change Number
E-1 through E-6	1
IN-1 and IN-2	0
IN-3 and IN-4	1
IN-5	0
IN-6	1
IN-7	0
IN-8 and IN-9	1
IN-10	0

* Zero in this column indicates an original page.

On a change page, the portion of the page affected by the latest change is indicated by a vertical bar in the outer margin of the page. However, a completely changed page will not have a full length bar, but will have the change notation by the page number.

CONTENTS

History . . . . .

CHAPTER 1 INTRODUCTION

1.1 Modes of Installation 1-1
1.2 FORTRAN 77+ Release 4.1 Versus Release 4.0 1-3
1.3 FORTRAN 77+ Versus ANSI X3.9-1978 1-3
1.4 Documentation Conventions 1-5

CHAPTER 2 THE FORTRAN PROGRAM

2.1 Program Units 2-1
2.1.1 Main Program 2-1
2.1.2 PROGRAM Statement 2-1
2.1.3 Subprogram 2-1
2.2 Fields 2-2
2.2.1 Statement Label Field 2-2
2.2.2 Continuation Field 2-2
2.2.3 Statement Field 2-3
2.2.4 Identification Field 2-3
2.3 Statements 2-3
2.3.1 Executable Statement 2-3
2.3.2 Nonexecutable Statement 2-3
2.3.3 Compiler Directive Statement 2-4
2.3.4 Order of Statements 2-4
2.3.5 Multiple Statement 2-5
2.4 Lines 2-5
2.4.1 Comment Lines 2-5
2.4.2 Trailing Comment 2-6
2.4.3 Initial Line 2-6
2.5 Character Set 2-6
2.5.1 Alphabetic Characters 2-6
2.5.2 Numeric Characters 2-7
2.5.3 Alphanumerics 2-7
2.5.4 Special Characters 2-7
2.6 Compiler Directive 2-7
2.6.1 INCLUDE Directive 2-7
2.6.2 OPTION Directive 2-9
2.6.3 PAGE Directive 2-10
2.6.4 SPACE Directive 2-11
2.6.5 USER Directive 2-11

CHAPTER 3 DATA TYPES, CONSTANTS, SYMBOLIC NAMES, VARIABLES, AND ARRAYS

3.1 General 3-1
3.2 INTEGER Data Types 3-1
3.2.1 Integer Constants 3-2

3.3 REAL Data Type 3-3
3.3.1 Real Constants 3-4
3.4 DOUBLE PRECISION Data Type 3-5
3.4.1 Double Precision Constants 3-5
3.5 COMPLEX Data Types 3-6
3.5.1 Complex Constants 3-7
3.6 Bit Data Type 3-8
3.7 LOGICAL Data Types 3-8
3.7.1 Logical or Bit Constants 3-8
3.8 CHARACTER Data Type 3-9
3.8.1 Character Constant 3-9
3.8.2 Character Constants versus Hollerith Constants 3-10
3.9 Hollerith Constants 3-10
3.10 Hexadecimal Constants 3-11
3.11 Binary Constants 3-12
3.12 Octal Constants 3-12
3.13 Symbolic Names 3-13
3.13.1 Implicit Typing Conventions 3-13
3.14 Variables 3-14
3.15 Arrays 3-14
3.15.1 Array Declarators 3-15
3.15.2 Dimensions of an Array 3-15
3.15.3 Subscripts 3-16
3.15.4 Array Storage 3-17

CHAPTER 4 EXPRESSIONS

4.1 General 4-1
4.2 Arithmetic Expressions 4-1
4.2.1 Evaluation of Arithmetic Expressions 4-6
4.3 Character Expressions 4-7
4.3.1 The Value of a Character Expression 4-8
4.3.2 Character Constant Expression 4-8
4.3.2.1 Character Substring 4-8
4.4 Relational Expressions 4-10
4.4.1 Relational Operators 4-10
4.4.2 Arithmetic Relational Expressions 4-10
4.4.3 Character Relational Expressions 4-11
4.5 Logical Expressions 4-12
4.5.1 Logical Operators 4-12
4.5.2 Evaluation of Logical Expressions 4-14
4.6 Use of Hollerith Constants in Expressions 4-15
4.7 Use of Hexadecimal, Binary, and Octal Constants in Expressions 4-17
4.8 Use of Strings in Argument Lists 4-18

CHAPTER 5 ASSIGNMENT STATEMENTS

5.1 General 5-1
5.2 Arithmetic Assignment Statements 5-2
5.3 Logical Assignment Statements 5-8
5.4 Character Assignment Statement 5-9
5.5 ASSIGN Statement 5-10
5.6 Multiple Assignment Statements 5-11
5.7 Full Array Assignments 5-12

CHAPTER 6 CONTROL STATEMENTS

6.1 General 6-1
6.2 GO TO Statements 6-2
6.2.1 Unconditional GO TO Statement 6-2
6.2.2 Computed GO TO Statement 6-3
6.2.3 Assigned GO TO Statement 6-4
6.3 IF Statements 6-5
6.3.1 Arithmetic IF Statement 6-5
6.3.2 Logical IF Statement 6-6
6.3.3 Block IF Construct 6-7
6.3.3.1 IF THEN Statement 6-8
6.3.3.2 ELSE IF THEN Statement 6-9
6.3.3.3 ELSE Statement 6-11
6.3.3.4 END IF Statement 6-12
6.3.3.5 Nested Block IF Construct 6-13
6.3.3.6 IF Level 6-14
6.4 DO Statement 6-15
6.4.1 Range of the DO Statement 6-16
6.4.2 Active and Inactive DO Loops 6-16
6.4.3 Terminal Statement of the DO 6-17
6.4.4 Index of the DO 6-17
6.4.5 DO Iteration Control 6-18
6.4.6 Nested DO Loops 6-18
6.4.6.1 Transfer of Control in Nested DO Loops 6-19
6.5 DO Forever Statement 6-20
6.6 DO UNTIL Statement 6-21
6.7 DO WHILE Statement 6-22
6.8 LEAVE Statement 6-23
6.9 CONTINUE Statement 6-24
6.10 END DO Statement 6-24
6.11 SELECT CASE Control Structure 6-25
6.11.1 SELECT CASE Statement 6-25
6.11.2 CASE Statement 6-26
6.11.3 END SELECT Statement 6-27
6.12 STOP Statement 6-28
6.13 END Statement 6-28
6.14 PAUSE Statement 6-29

CHAPTER 7 SPECIFICATION STATEMENTS

7.1 Introduction 7-1
7.2 DIMENSION Statement 7-2
7.3 Type Statements 7-3
7.3.1 Explicit Type Statement 7-4
7.3.2 Explicit CHARACTER Statement 7-6
7.3.3 IMPLICIT Statement 7-8
7.3.4 IMPLICIT NONE Statement 7-9
7.4 PARAMETER Statement 7-10
7.5 COMMON Statement 7-11
7.5.1 Global COMMON 7-13
7.5.2 DATAPOOL 7-14
7.6 EQUIVALENCE Statement 7-15
7.6.1 EQUIVALENCE and Boundaries 7-16
7.6.2 EQUIVALENCE and Arrays 7-17

7.6.3 EQUIVALENCE and Substrings 7-20
7.6.4 EQUIVALENCE and COMMON Interaction 7-22
7.7 EXTERNAL Statement 7-23
7.8 INTRINSIC Statement 7-25
7.9 SAVE Statement 7-26

CHAPTER 8 DATA STATEMENT

8.1 General 8-1
8.2 Implied DO in a DATA Statement 8-4

CHAPTER 9 FUNCTIONS AND SUBROUTINES

9.1 General 9-1
9.1.1 Dummy Arguments 9-1
9.2 Statement Functions 9-3
9.2.1 Referencing a Statement Function 9-4
9.3 Intrinsic Functions 9-5
9.3.1 Specific Names and Generic Names 9-6
9.3.2 Inline Intrinsic Functions 9-6
9.4 Function Subprogram 9-19
9.4.1 Referencing a Function Subprogram 9-20
9.5 Subroutine Subprograms 9-21
9.5.1 Referencing a Subroutine Subprogram 9-22
9.6 Internal Procedures (Function and Subroutine) 9-24
9.6.1 Dummy Arguments 9-25
9.6.2 Referencing Internal Procedures 9-25
9.7 ENTRY Statement 9-26
9.7.1 ENTRY Association 9-27
9.8 Function and Subroutine Subprograms Returns 9-27
9.8.1 Alternate Returns 9-28
9.9 Processing Arrays in Subprograms 9-30
9.10 Processing of Arguments for Subprogram Calls 9-31
9.11 Mismatched Argument Lists 9-32

CHAPTER 10 BLOCK DATA SUBPROGRAMS

10.1 Introduction 10-1
10.2 BLOCK DATA Statement 10-1

CHAPTER 11 INPUT/OUTPUT

11.1 Introduction 11-1
11.2 Records 11-1
11.3 Files 11-2
11.4 Sequential and Direct Access Methods 11-3
11.5 Blocked and Unblocked Files 11-3
11.6 Control Information List 11-4
11.6.1 The Unit Specifier 11-5
11.7 Input/Output List 11-5
11.7.1 Single Datum Identifier 11-6
11.7.2 Multiple Data Identifiers 11-6
11.8 Control and Interpretation of Data 11-7

11.9 Input/Output Statements 11-8
11.9.1 Input Statements 11-8
11.9.2 Output Statements 11-13
11.9.3 Input and Output Using NAMELIST 11-19
11.9.3.1 Input from a User Terminal 11-20
11.9.3.2 Input from Other Than a User Terminal 11-20
11.9.3.3 Input Data Item Format 11-21
11.9.3.4 Output Data Formats 11-22
11.10 List-Directed Formatting 11-24
11.10.1 List-Directed Input 11-25
11.10.2 List-Directed Output 11-26
11.11 Auxiliary Input/Output Statements 11-27
11.11.1 OPEN Statement 11-27
11.11.2 CLOSE Statement 11-39
11.11.3 INQUIRE Statement 11-41
11.11.3.1 INQUIRE by File, Native Mode 11-41
11.11.3.2 INQUIRE by File, Compatible Mode 11-41
11.11.3.3 INQUIRE by Unit, Native Mode 11-42
11.11.3.4 INQUIRE by Unit, Compatible Mode 11-43
11.11.3.5 INQUIRE Statement Specifiers 11-43
11.11.4 BACKSPACE Statement 11-51
11.11.5 BACKFILE Statement 11-52
11.11.6 SKIPFILE Statement 11-53
11.11.7 ENDFILE Statement 11-54
11.11.8 REWIND Statement 11-56
11.12 ENCODE and DECODE Statements 11-57
11.13 BUFFERIN and BUFFEROUT (Asynchronous I/O) 11-59
11.14 CALL STATUS 11-62

CHAPTER 12 FORMAT SPECIFICATION

12.1 General 12-1
12.2 Format Specification Methods 12-1
12.2.1 FORMAT Statements 12-1
12.2.2 Format Specifications Stored in Variables and Arrays 12-2
12.2.3 Format Specifications Expressed as Character Constants 12-2
12.3 Form of a Format Specification 12-3
12.3.1 Edit Descriptors 12-3
12.3.2 Interpretation of Blanks on Input 12-4
12.4 Format Control List Specifications and Record Demarcation 12-5
12.5 Forms Control on Output 12-6
12.6 Numeric Editing 12-7
12.6.1 D Editing and Output 12-7
12.6.2 D Input 12-8
12.6.3 E Editing 12-9
12.6.4 E Output and Input 12-9
12.6.5 F Editing 12-10
12.6.6 F Output and Input 12-10
12.6.7 G Editing 12-10
12.6.8 G Output and Input 12-11
12.6.9 Complex Editing 12-12
12.6.10 I Editing 12-12

12.6.11 I Output 12-12
12.6.12 I Input 12-13
12.6.13 Z Editing 12-13
12.6.14 Z Output 12-13
12.6.15 Z Input 12-14
12.6.16 S, SP, and SS Descriptors 12-14
12.6.17 BN and BZ Descriptors 12-15
12.7 Character Editing 12-16
12.7.1 A Editing 12-16
12.7.2 A Output 12-16
12.7.3 A Input 12-17
12.7.4 Apostrophe Editing 12-17
12.7.5 H Editing 12-17
12.7.6 H Output 12-17
12.7.7 A Editing with Noncharacter Data Types 12-18
12.7.8 A Output with Noncharacter Data Types 12-18
12.7.9 A Input with Noncharacter Data Types 12-18
12.7.10 R Editing 12-19
12.7.11 R Output 12-19
12.7.12 R Input 12-19
12.8 Logical Editing 12-20
12.8.1 L Output 12-20
12.8.2 L Input 12-20
12.9 Positional Editing 12-21
12.9.1 The X Descriptor 12-21
12.9.2 The T Descriptor 12-21
12.10 Scale Factors 12-22
12.11 Repeat Specifications 12-24
12.12 Field Separators 12-26
12.13 Colon Descriptor 12-27

CHAPTER 13 EXTENDED ADDRESSING

13.1 Introduction 13-1
13.2 EXTENDED BLOCK Statement 13-1
13.3 EXTENDED BASE Statement 13-2
13.4 Extended Dummy Statement 13-2
13.5 Extended Memory Restrictions 13-4

CHAPTER 14 INLINE ASSEMBLY LANGUAGE CODING

14.1 Introduction 14-1
14.2 Label Field 14-1
14.3 Operation Field 14-1
14.4 Argument Field 14-1
14.5 Comment Field 14-1
14.6 General Instruction Format 14-2
14.6.1 Memory Reference Instructions 14-2
14.6.2 Interregister Instructions 14-3
14.6.3 Immediate Operand Instructions 14-3
14.6.4 Memory Bit and Condition Code Instructions 14-3
14.6.5 Operation Control Instructions 14-4
14.7 Argument Field Conventions 14-4
14.8 DATA Directive 14-4
14.9 GEN Directive 14-5

14.10 AC Directive 14-5
14.11 BOUND Directive 14-5
14.12 RES Directive 14-6
14.13 EQU Directive 14-6
14.14 Referencing Variables in Local Storage, COMMON, GLOBAL COMMON, or DATAPOOL 14-6
14.15 Referencing Dummy Variables 14-6
14.16 Referencing Variables in Extended Memory 14-7
14.17 Setting and Clearing Extended Addressing Mode 14-7

CHAPTER 15 USING THE FORTRAN 77+ COMPILER

15.1 Introduction 15-1
15.2 Logical File Code Assignments 15-1
15.3 Compiler Options 15-1
15.4 Run-Time Options 15-3
15.5 Job Control Language 15-3
15.5.1 Compiling 15-4
15.5.2 Compiling, Cataloging, and Executing 15-4
15.5.3 Compiling and Cataloging 15-5
15.6 Using a Tape File as a Data Source 15-5
15.7 Using a Disc File for Output Data 15-6
15.8 Using Data from Cards 15-7
15.9 Calling Assembly Routines from FORTRAN 77+ Programs 15-7
15.9.1 Assembly Routine with No Parameters 15-8
15.9.2 Assembly Routine with One Parameter 15-8
15.9.3 Assembly Routine with Two or More Parameters 15-8
15.9.4 Parameter Area 15-8
15.9.5 Calculation of the Return Address 15-9
15.9.6 Function Calling Conventions 15-17
15.10 Calling FORTRAN 77+ Subroutines from Assembly Language Programs 15-17
15.10.1 FORTRAN 77+ Subroutine with No Parameters 15-17
15.10.2 FORTRAN 77+ Subroutine with One Parameter 15-17
15.10.3 FORTRAN 77+ Subroutine with Two or More Parameters 15-18
15.10.4 Parameter Lists Generated by the Compiler 15-18

APPENDIX A I/O USING MPX-32

A.1 Input/Output Terms A-1
A.2 General Observations A-2
A.3 End-of-File Detection A-2
A.4 Maximum Sizes A-2
A.5 Unformatted I/O Records A-3
A.6 Device Type Codes A-4

APPENDIX B LISTING EXAMPLES

B.1 Source Listing B-1
B.1.1 Source Listing Format B-1
B.2 Storage Dictionary B-9
B.2.1 Storage Dictionary Format B-9
B.3 Symbolic Cross Reference B-13
B.3.1 Symbolic Cross Reference Format B-13

B.4 Generated Code Listing B-9
B.4.1 Generated Code Listing Format B-9

APPENDIX C ASCII CODE SET C-1

APPENDIX D DIAGNOSTICS

D.1 Compile-Time Diagnostics D-1
D.1.1 Source Line Errors D-1
D.1.2 Context Errors D-1
D.1.3 Abort Codes - FT D-3
D.2 Execution-Time Diagnostics D-4A
D.2.1 IOSTAT Values D-5
D.2.2 Abort Codes - RS and RT D-11
D.3 Minor Errors D-18

APPENDIX E COMPARISON OF FORTRAN 77+ AND FORTRAN 77/X32 . . . . . E-1

INDEX IN-1

FIGURES

Figure

2-1 Order of Statements 2-4
3-1 Array Storage 3-18
6-1 Simple Block IF Construct 6-8
6-2 Block IF Construct with ELSE IF THEN Statement Block 6-9
6-3 Block IF Construct with Multiple ELSE IF THEN Statement Blocks 6-10
6-4 Block IF Construct with ELSE Statement Block 6-11
6-5 Nested DO Loops 6-18
7-1 Mixed Data Type Storage in Common 7-12
7-2 Equivalence of Character Arrays 7-21

TABLES

Table

1-1 FORTRAN 77+/SRTL Installation Modes 1-2
4-1 Expression Type Determination from Arithmetic Operands of Different Types 4-4
4-2 Hollerith Constants Used with Operators 4-15
4-3 Hollerith or Hexadecimal String Conventions 4-18
7-1 Standard and Optional Type Lengths 7-3
7-2 Equivalence of Array Storage 7-18
7-3 Equivalence of Arrays with Nonunity Lower Bounds 7-19
9-1 Arithmetic and Conversion Intrinsic Functions 9-7
9-2 Lexical Comparison Intrinsic Functions 9-14
9-3 Word and Bit Intrinsic Functions 9-15
9-4 Trigonometric Intrinsic Functions 9-17
11-1 Input Statements 11-11
11-2 Output Statements 11-16
11-3 Maximum Transfer Counts (E Class) 11-60
15-1 Compiler Parameter Lists 15-19
15-2 Type Code 15-20

CHAPTER 1

INTRODUCTION

This manual describes the Gould Model 1413-2 FORTRAN 77+ compiler, which supports a superset of ANSI X3.9-1978 when combined with the Gould Scientific Run-Time Library. FORTRAN 77+ also complies with MIL-STD-1753 and ISA Standards S61.1 and S61.2.

1.1 Modes of Installation

There are four possible installation modes for the FORTRAN 77+ compiler. They are: native/hardware assisted, native/non-hardware assisted, compatible/hardware assisted, and compatible/non-hardware assisted.

The native and compatible aspects of the compiler are indicative of the file system interface being selected with regards to the Scientific Run-time Library (SRTL). Code for accessing available MPX-32 resources, when generated by the native mode compiler, will call SRTL routines which access the MPX-32 file system in accordance with methods native only to MPX-32 Rev. 2 and Rev. 3. Similar code, when generated by the compatible mode compiler, will call SRTL routines which access the MPX-32 file system in accordance with methods compatible with MPX-32 Rev. 1. The installation of this release of the compiler is not supported on MPX-32 Rev. 1. However, FORTRAN source code developed under prior releases of FORTRAN 77+ running under MPX-32 Rev. 1, will still be valid if compiled in the compatible mode.

The hardware assisted and non-hardware assisted aspects of the compiler allow for the selection or non-selection of certain hardware floating point instructions. These are available on CONCEPT 32/67, 32/87 and 32/97 machines. Code generated by the compiler in the hardware assisted mode will contain special hardware FIX/FLOAT register to register instructions whenever conversion from floating point math to fixed point or vice versa is required. Code generated in the non-hardware assisted mode will contain calls to SRTL floating point and fixed point conversion routines when such conversion requirements exist.

The compiler can be used in the default mode in which it was installed, or it can optionally be used in other modes. If the compiler is installed with the native mode as default, compatible mode can still be selected by setting TSM option 12 at the beginning of compilation. Conversely, option 17 allows the selection of native mode from a compiler which was installed with compatible mode as the default. If both options 12 and 17 are used, the compiler will default to compatible mode no matter what the installation default mode was. Option 13 will allow a compiler installed with hardware assisted mode as its default to run in the non-hardware assisted mode. There is no option to allow a compiler installed in the non-hardware assisted mode to run in the hardware assisted mode. Chapter 15 describes the compiler options.

In order to catalog programs compiled under FORTRAN 77+, the SRTL must be assigned as one of the available object libraries to the cataloger. Selection of the mode for the SRTL can not be changed by options but is solely dependent upon which installation of

the SRTL is assigned to the cataloger. The SRTL has the same possible installation modes as does the compiler. It is therefore recommended that the modes selected at installation for the SRTL, match those selected for the compiler installation.

Table 1-1
FORTRAN 77+/SRTL Installation Modes

		REV. 2	REV. 3
SERIES 32/7X	NON-HW ASSIST ONLY	compatible native *	
CONCEPT 32/27	NON-HW ASSIST ONLY	compatible native	compatible native *
CONCEPT 32/67	HW ASSIST		compatible native *
	NON-HW ASSIST		compatible native
CONCEPT 32/87	HW ASSIST	compatible native	compatible native *
	NON-HW ASSIST	compatible native	compatible native
CONCEPT 32/97	HW ASSIST		compatible native *
	NON-HW ASSIST		compatible native
* - RECOMMENDED			

Care must be taken at all times, to ensure that the modes used during compilation match the modes of the SRTL used during cataloging. The programming aspects that are mode dependent are so indicated in this manual. For a complete description of the file systems, refer to the MPX-32 Reference Manual. The SRTL Reference Manual further describes the relationship between the SRTL and the compiler.

1.2 FORTRAN 77+ Release 4.1 Versus Release 4.0

Release 4.1 of the FORTRAN 77+ compiler differs from Release 4.0 in the following ways:

- The compiler may optionally use big blocking buffers for blocked I/O as defined by the operating system on which the compiler is running.
- Regular FORMAT statements are now converted to format item tables. These are readily used by the run-time library routines to eliminate redundant re-evaluation of formats during run time I/O.
- The overlay structure of the compiler has been replaced by sectioned code to make the compiler a shared task. Symbol tables and work stacks are now in a 32K word area of extended memory. This completely eliminates the need for, and prohibits the use of, the $ALLOCATE directive at compiler execution time.
- Extended memory addressing has been improved.
- The logical functions IAND, IOR, IEOR, and NOT will now be expanded inline unless these functions are specifically defined as external to the program.
- Date and time of compilation and product identification information is now collected at the user's option and stored in the object code and load module.
- A second conditional compilation marker, Y, has been added to the already available marker, X.
- The utility of BUFFERIN/BUFFEROUT has been modified so that the user can select a "sector specifier" as an optional parameter in order to do random I/O on disc files only.
- Modifications were made to allow FORTRAN 77+ to resolve bit variables in DATAPOOL items.
- Identification of file system mode and hardware assist and/or Scientific Accelerator 67 options are listed on the title output line.

1.3 FORTRAN 77+ Versus ANSI X3.9-1978

The full language of ANSI X3.9-1978 is supported. However, FORTRAN 77+ differs from it in the following ways:

- Assembly language statements can be intermixed with FORTRAN 77+ statements.
- The first four characters of a keyword (or the entire keyword if fewer than four characters) must be on a single line. For IF statements, the subsequent opening parenthesis is considered part of the keyword.
- The receiving operand and the equal sign of an assignment (replacement) statement must appear on the same line. This line must be the first line of the statement unless the assignment statement follows a logical IF. In that case, the replacement must occur no later than the end of the first line after the end of the logical expression or closing parenthesis.

- The equal sign and first comma of a DO statement must appear on the initial line of the statement.
- Symbolic names can be of unlimited length. However, only the first eight characters are significant in establishing identity. The colon (:) and underline (_) characters are permitted in names.
- Statements are not limited to 19 continuation lines.
- Hexadecimal, binary, octal, and Hollerith constants are permitted in DATA statements.
- Multiple statements per line, separated by semicolons, are permitted.
- Multiple assignment statements are provided.
- Additional format codes (Z and R) are provided.
- An additional form of Hollerith constant (nR) is provided.
- Hexadecimal, binary, and octal constants (Z and X, B, and O forms) are provided.
- Subscript expressions are converted (truncated) to integer values.
- Data type specification statements (implicit or explicit) can specify a size for each variable as follows:

  Integer byte (INTEGER*1)
  Integer halfword (INTEGER*2)
  Integer word (INTEGER or INTEGER*4)
  Integer doubleword (INTEGER*8)
  Real (REAL or REAL*4)
  Double precision (DOUBLE PRECISION or REAL*8)
  Complex word (COMPLEX or COMPLEX*8)
  Complex doubleword (COMPLEX*16)
  Logical bit (BIT)
  Logical byte (LOGICAL*1)
  Logical word (LOGICAL or LOGICAL*4)
- Explicit data type specification statements can include data initialization.
- The capability of assigning a value to a full array with a single assignment statement is provided.
- The IMPLICIT NONE statement is provided.
- DO statement extensions such as DO forever, DO WHILE, DO UNTIL, END DO, and LEAVE are provided.
- SELECT CASE, CASE, CASE DEFAULT, and END SELECT are provided.
- INTERNAL procedure statements are provided.
- Additional input/output features are provided:

  Asynchronous input/output (BUFFERIN and BUFFEROUT)

PUNCH, TYPE, and ACCEPT statements with implied unit designators (file codes)

ENCODE and DECODE compatible with FORTRAN IV

Namelist READ/WRITE

BACKFILE and SKIPFILE

I/O unit status availability by means of STATUS and EOF subroutines

- Additional comment features are provided:

  End-of-line comments initiated by an exclamation mark (!)

  Optional compilation of X and Y comment lines

- No check is made at compile time or execution time on the type of arguments supplied in subroutine calls or function references.

- No checks are made on the values of array subscripts, except in EQUIVALENCE statements.

- The format specifier in a formatted READ or WRITE statement can be a variable, array element, or array name with a data type other than CHARACTER.

- The unit specifier in an input/output statement can be a logical file code in the form 'lfc', where 'lfc' consists of from 1 to 3 ASCII characters.

- Statements are provided for using the extended memory feature (EXTENDED BLOCK, EXTENDED BASE, EXTENDED DUMMY).

- USER, INCLUDE, SPACE, PAGE, and OPTION directives are provided.

1.4 Documentation Conventions

The conventions followed in documenting command syntax, examples, and messages in this manual are:

- **lowercase letters** identify a variable element that must be replaced with a value. For example,

  PROGRAM name

  means you must enter the symbolic name of the main program in which the PROGRAM statement appears.

- UPPERCASE LETTERS specify a string that must be entered as shown for input, and is printed as shown in output. For example,

  PROGRAM name

  means enter PROGRAM followed by the name.

- **Brackets** surrounding an element specify it as optional. For example,

  DO [x] [,] UNTIL (e)

  means x and the comma (,) are optional.

- **Braces** surrounding elements specify a required choice. You must enter one of the arguments from the group. For example,

  OPTION n_1 { ± }

  means enter either a plus sign (+) or a minus sign (-).

- **Parentheses** are part of the command line typed by the user and must appear as shown in statement syntax when input. For example,

  IF (e) THEN

  means an expression e is required and must be enclosed by parentheses.

- **Horizontal ellipsis** following an element means the element can be repeated as many times as needed. For example,

  CASE $const_1$ [,$const_2$] . . .

  means enter one or more constant expressions separated by commas.

- **Vertical ellipsis** specifies that commands, parameters, or instructions are omitted. For example,

```
J=3
 .
 .
 .
GO TO 115
```

  means one or more commands are omitted between the J=3 and the GO TO 115 commands.

- ␣ specifies a blank. For example,

  ␣&NAM1␣I(2,3)=5,

  means &NAM1 must be preceded and followed by a blank.

- A **carriage return** is required at the end of every input line in programming examples.

CHAPTER 2

THE FORTRAN 77+ PROGRAM

2.1 Program Units

A FORTRAN 77+ program is composed of one or more program units. A program unit consists of a sequence of statements and optional comment lines. A program unit is either a main program or a subprogram. The last line of a program unit is denoted by an END statement.

2.1.1 Main Program

A main program is a program unit that does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement; it can have a PROGRAM statement as its first statement.

A FORTRAN 77+ program must have one and only one main program. A main program can not be referenced by a CALL statement within another program unit.

2.1.2 PROGRAM Statement

A PROGRAM statement is not required. If used, it must be the first statement of the main program.

Syntax

PROGRAM name

name — The symbolic name of the main program in which the PROGRAM statement appears. This name may not be used as a variable name elsewhere in the main program.

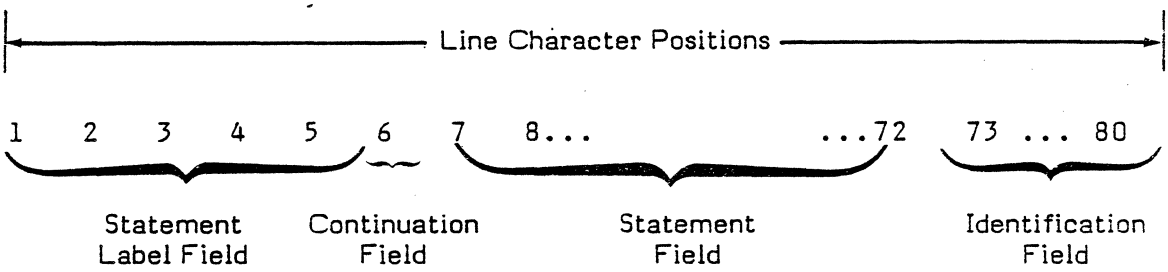
The name MAIN is assigned to a main program that does not contain a PROGRAM statement.

2.1.3 Subprogram

A subprogram is a program unit that has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

2.2 Fields

Each line of a FORTRAN 77+ program is divided into four fields.



2.2.1 Statement Label Field

A statement label is a numerical means of referring to an individual line of a FORTRAN 77+ program. The statement label field is character positions 1 through 5.

The following rules govern the use of statement labels:

- A label is an integer from 1 to 99999. Leading zeros and leading or embedded blanks in the label are not significant. Each of the following statement labels is the number 102.

```
1  2  3  4  5...
      1  0  2
1  0        2
0  0  1  0  2
```

- A label is placed on the initial line of a statement.
- A label is unique within a program unit.
- Any statement can be labeled, but only executable and FORMAT statements can be referenced by the use of statement labels.

2.2.2 Continuation Field

The continuation field, character position 6, indicates that the line is a continuation line. Continuation lines are used when a statement is too long to fit on one line.

The following rules govern the use of continuation lines:

- Character positions 1 through 5 are blank.
- Character position 6 contains any FORTRAN 77+ character except blank or zero.
- Character positions 7 through 72 contain the continuation of the statement.
- There is no limit on the number of continuation lines used.

Example

```
1   2   3   4   5   6   7 8                                   ... 72
                        Y = (A*X**2+B**2+
                    +   C - (A**2+B**2))/
                    +   ( A -B-C)
```

2.2.3 Statement Field

The statement field--character positions 7 through 72--contains assignment, specification, control, input/output, or function statements. These statements are composed of keywords--such as DO and GO TO--along with constants, variables, operators, and delimiters.

The first four letters of a keyword must be on the same line. For example, GO TO of a GO TO statement and SELE of a SELECT CASE statement. Furthermore, the left parenthesis of an IF statement must be on the same line as the keyword IF.

Blanks have no meaning in a statement field; therefore, they can be used to make statements more legible.

2.2.4 Identification Field

The identification field--character positions 73 through 80--is used for comments or other input that is not to be processed. The compiler ignores the field.

2.3 Statements

FORTRAN 77+ statements are composed of keywords used with constants, variables, operators, and delimiters that are elements of the language set. FORTRAN 77+ statements are either executable, nonexecutable, or compiler directives.

2.3.1 Executable Statement

An executable statement specifies actions. There are three types of executable statements:

- Assignment
- Control
- Input/Output

2.3.2 Nonexecutable Statement

A nonexecutable statement specifies the nature and arrangement of data. There are six types of nonexecutable statements:

- Specification
- Data initialization
- Statement function definition
- Subprogram
- Program
- Format or namelist

2.3.3 Compiler Directive Statement

A compiler directive statement provides information required for compilation. Compiler directives are described in Section 2.6. These statements are:

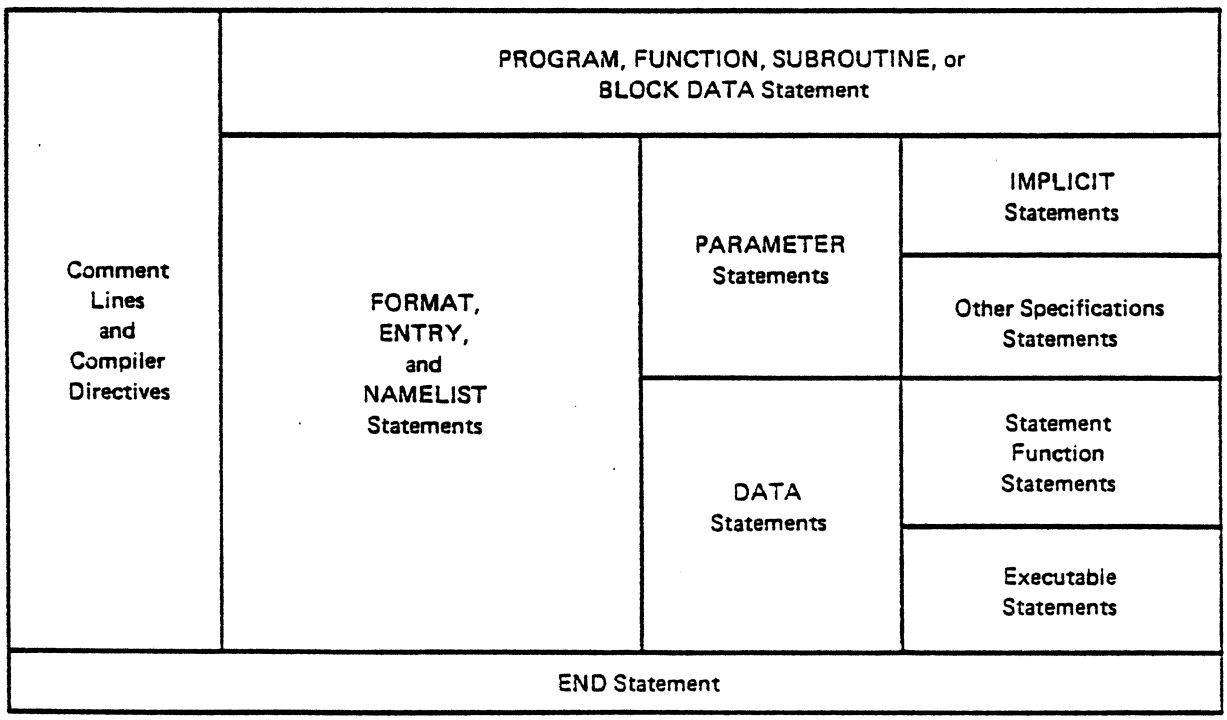
- INCLUDE
- OPTION
- PAGE
- SPACE
- USER

2.3.4 Order of Statements

Statements must be written in a certain order so that compilation proceeds as expected. The order of statements and comments for FORTRAN 77+ is summarized in Figure 2-1.

Horizontal lines in Figure 2-1 indicate the order in which statements appear. For example, PARAMETER statements follow BLOCK DATA statements, but they precede DATA statements. Statement function statements follow IMPLICIT and other specification statements, but they precede executable statements.

Vertical lines in Figure 2-1 indicate the way in which statements can be interspersed. For example, FORMAT, ENTRY and NAMELIST statements can be interspersed with PARAMETER, DATA, statement function, and executable statements; and with IMPLICIT and other specification statements.



810552A

Figure 2-1. Order of Statements

2.3.5 Multiple Statement

A multiple statement consists of several statements on the same line, with the statements separated by semicolons. A label on a line with multiple statement applies to the first statement on the line.

Example

```
10    I=N/4; J=N*M+4; X=Y(I,J)
```

is the same as

```
10    I=N/4
      J=N*M+4
      X=Y(I,J)
```

2.4 Lines

In addition to executable, nonexecutable, and compiler directive statements, FORTRAN programs can also contain comment lines.

2.4.1 Comment Lines

Comment lines are explanatory notes within a program that make the program more understandable to a programmer. Comment lines are ignored by the compiler.

The following rules govern the use of comment lines:

- Character position 1 has a C, *, X, or Y. Character positions 2 through 72 can contain any character.
- When option 20 is set, a line with an X in character position 1 is processed as a FORTRAN 77+ statement.
- When option 9 is set, a line with a Y in character position 1 is processed as a FORTRAN 77+ statement.
- A line with blanks in character positions 1 through 72 is a comment line.

Example

```
1 2 3 4 5 6 7 8 . . .                                    . . . 72
*       THIS  IS  A  COMMENT  LINE
C       NOTE THE CHARACTERS C AND * IN
C       COLUMN 1
X       PRINT  100,  A,  B
X 1 0 0 FORMAT  (1H1,  'DEBUG',  2F10.3)
C       THE  ABOVE  TWO  STATEMENTS  ARE
C       TREATED  AS  COMMENTS  UNLESS
C       OPTION  2 0  IS  SET
Y       DO  I=1,10
Y         X(I)=Y(I)+Z(I)
Y       END  DO
C       THE  ABOVE  THREE  STATEMENTS  ARE
C       TREATED AS COMMENTS UNLESS
C       OPTION  9  IS  SET.
```

2.4.2 Trailing Comment

Comments added to the end of a statement are trailing comments. An exclamation point (!) anywhere within character positions 7 through 72, except in character constants, marks the beginning of a trailing comment. The trailing comment can continue through character position 72 of that line.

Example

```
1 . . .  6 7 . . .                                          . . .72
      A=B*(C+D)  ! THIS IS A COMMENT
```

2.4.3 Initial Line

An initial line is the first line of a statement. The following rules govern the use of initial lines:

- Character positions 1 through 5 may contain a statement label or each of the character positions 1 through 5 must contain a blank.
- Character position 6 is a blank or zero.

Refer to section 1.3 for additional information.

Example

```
1   2   3   4   5   6   7  . . .                          . . .72
1   0   3                  ALPH= .5 + SIN(B1) - SIN(C2)
2   0   0                  A = B * (C + D)
2   2   0                  IF (A. GT. 10 .OR. B .LE.  P/2)
                     X     I = J
                           DO 330 M = 40,
                     X     60, 2
                           .
                           .
                           .
```

2.5 Character Set

The following sections contain the characters which may be used in writing FORTRAN 77+ programs.

2.5.1 Alphabetic Characters

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

Lowercase ASCII letters are converted to uppercase except in Hollerith and character strings. The order in which the letters are listed above specifies the collating sequence; that is, A is less than Z, Z is less than a, and a is less than z.

2.5.2 Numeric Characters

0,1,2,3,4,5,6,7,8,9

The order in which the digits are listed above specifies the collating sequence; that is, 0 is less than 9.

2.5.3 Alphanumerics

Alphanumeric characters include all alphabetic and numeric characters, and the characters colon (:) and underline (_).

2.5.4 Special Characters

	Blank	;	Semicolon
!	Exclamation point	<	Less than
"	Quotation mark	=	Equal sign
#	Number sign	>	Greater than
$	Dollar sign	?	Question mark
%	Percent sign	@	At sign
&	Ampersand	[	Opening bracket
'	Apostrophe	\	Backslash
(	Opening parenthesis	]	Closing bracket
)	Closing parenthesis	^	Circumflex
*	Asterisk	_	Underline
+	Plus sign	`	Accent grave
,	Comma	{	Opening brace
-	Minus sign	|	Vertical line
.	Decimal point	}	Closing brace
/	Slash	~	Tilde
:	Colon		

The order in which the special characters are listed above specifies the collating sequence; that is, ! is less than $, and : is less than ;.

2.6 Compiler Directives

Compiler directives aid in listing control, selection of input from alternate files, and dynamic changing of options during compilation. Syntactically, directives are treated as comment lines. Directives cannot have continuation lines.

2.6.1 INCLUDE Directive

The INCLUDE directive is used to enter a portion of text from a named file with the source to be compiled. The compiler processes the INCLUDE directive from the current file, processes the source from the file specified in the INCLUDE directive, and continues processing after the INCLUDE directive. Nesting of INCLUDE directives within included source is permitted. The keyword INCLUDE must be within character positions 7 to 72.

Syntax

INCLUDE 'pathname' [,start, end]

INCLUDE filename [, [password],[start, end]]

'pathname' The pathname, enclosed in apostrophes, in the following form:

{ resource_name
[^] (directory_name)resource_name
@volume_name [^] (directory_name) resource_name }

volume_name — The 1 to 16 character volume name on which the required resource resides.

directory_name — The 1 to 16 character directory name on the specified volume in which the required resource is defined.

resource_name — The 1 to 16 character name of a permanent disc file to be located in the specified directory on the specified volume. It must contain source text in blocked, uncompressed form.

filename — The 1 to 8 character name of a permanent disc file containing source text in blocked, uncompressed form. The default is the current user (directory). If none is found, a system file of the same name will be included. If a system file is to be used and a user file by the same name exists, a USER directive with a blank user name must precede the INCLUDE directive.

password — Ignored. (Permitted for compatibility.)

start — The beginning line number of the text to be included within the named file. This is a physical line count, not a sequence number in character positions 73 through 80.

end — The final line number within the named file of the text to be included.

Rules for Use

- Either filename or pathname can be used in the INCLUDE directive. However, if filename is used, the file will be allocated from the current working volume and directory if it exists there. Otherwise, an attempt will be made to allocate it from the system volume and directory.
- Pathname can be used to include files in volumes and directories other than the current working volume and directory. This eliminates the need for the USER directive and removes the restrictions associated with INCLUDE filename. Refer to the MPX-32 Reference Manual for a complete description of pathname.
- If either start or end is used, both must appear. If using INCLUDE filename they must be preceded by two commas if no password is used. However, if an END statement is encountered before the end line number is reached, inclusion of text is terminated and a warning message is issued.

- If start and end are not specified, the entire text of the named file will be included for compilation. Text following an END statement will not be included.
- An INCLUDE statement cannot have a continuation line.
- No other statement may appear in the same line as an INCLUDE statement.
- Some special characters cannot be used as part of filename or pathname since they are delimiters. These include comma (,), at sign (@), parentheses (( )), and braces ({ }).

Examples

```
INCLUDE BIG_FILE,,1,10
INCLUDE '@LANG_2(PROJECT4) NOTES',5,15
INCLUDE ' ^(FRUIT) APPLE'
INCLUDE MYFILE,, 11,20
```

The portions of text entered by the preceding directives could be complete program units.

2.6.2 OPTION Directive

The OPTION directive initiates or suppresses specific compiler options within a program. Refer to Chapter 15 for a list of compiler options and the rules concerning the use of those options.

Syntax-1

OPTION n_1 { $\pm$ } [,n_2 { $\pm$ } . . .,n_i { $\pm$ }]

n_i Valid option numbers

OPTION n+ Turns option n on
OPTION n- Turns option n off

Example

```
OPTION 1+, 2+, 6-
```

The preceding example results in the following:

1+ Initiates option 1, which suppresses listed output
2+ Initiates option 2, which suppresses binary output
6- Turns off option 6, which lists generated code

Syntax-2

OPTION (DATE_TIME= { $\pm$ })

Example

```
OPTION (DATE_TIME=+)
```

Rules for Use

- OPTION (DATE_TIME=+) corresponds to the setting of TSM $OPTION 15.
- The date and time are placed in the module's object code and can be recognized by the cataloger using cataloger option 15. Note that the cataloger will not process this information for load modules containing overlays.

- The latest setting of OPTION 15 (DATE_TIME) is in effect for the remainder of the compilation unit, until changed by a source option (DATE_TIME = { + }) statement.

Syntax-3

OPTION (PRODUCT_ID = 'module identifier')

module identifier ASCII string of up to 32 characters

Example

OPTION (PRODUCT_ID='This is the product identifier')

Rules for Use

- The module identifier is placed in the module's object code and can be recognized by the cataloger by using cataloger option 15. Note that the cataloger will not process this information for load modules containing overlays.
- If more than one module identifier is specified in a program unit, only the first one is taken and an error message is issued.
- PRODUCT_ID is only in effect for the current program unit.

2.6.3 PAGE Directive

The PAGE directive causes the compiler's listed output to go to the top of the form. A heading may be specified for the top of each page of listed output. The PAGE directive itself is not in the listed output.

Syntax

PAGE [heading]

heading 1 to 62 ASCII characters.

Rules for Use

- A heading, if specified, appears at the top of each subsequent page in the listed output of the current program unit until changed by another PAGE directive containing a heading. A PAGE directive without a specified heading does not change the current heading.
- Consecutive PAGE directives without intervening text or blank lines will generate one page only.

Example

```
      PAGE DO LOOP STRUCTURE
      DO 20 NUMBER = I,J
      M = NUMBER + K
20    D(M) = Q (M)
        .
        .
        .
      PAGE IF STRUCTURE
80    IF (I.GT.20) GO TO 115
        .
        .
        .
      PAGE
103   ALPH = .5
        .
        .
        .
```

Note that the last PAGE directive will not result in a heading change.

2.6.4 SPACE Directive

The SPACE directive inserts a specified number of blank lines in a compiler listing. If the output is to other than SLO or UT, and fewer than n blank lines remain at the bottom of the page, blank lines will not be inserted at the top of the next page. The SPACE directive itself is not in the listing.

Syntax

```
SPACE [n]
```

n An integer constant. The default for n is one.

Examples

```
SPACE 4     ! Four blank lines will be placed at this point in the listed output.

SPACE       ! One blank line will be placed at this point in the listed output.
```

2.6.5 USER Directive

The USER directive is required when selecting a file with the INCLUDE file directive indicating the directory to be searched.

Syntax

```
USER [username [,key]]
```

username A sequence of 1 to 8 ASCII characters that redefines the directory.

key Ignored. (Permitted for compatibility.)

Rules for Use

- The USER specified remains in effect until another USER directive is specified (unless within an included file).
- A USER directive within an included file only applies to the next INCLUDE filename directive.
- The USER directive with no username (e.g., USER ...) establishes the SYSTEM directory, if one exists, on the current working volume as a default.
- The USER directive cannot be used to change to a username on a different volume.

Example

```
USER JOHNB, CD          ! The USER name JOHNB becomes the username in
                        ! effect for this task. The key, CD, is ignored.
```

CHAPTER 3

DATA TYPES, CONSTANTS, SYMBOLIC NAMES, VARIABLES, ARRAYS

3.1 General

Symbolic names identify data items. Each symbolic name must have a data type; constants always have a data type. The data type defines the operations that can be performed on the data.

The seven data types are:

- integer
- real
- double precision
- complex
- bit
- logical
- character

3.2 Integer Data Types

Integer data types are used to represent exact integer values. The four integer data types differ in the range of values that each can represent and the number of bytes required for storage (one storage unit = one byte).

How Specified in Type Specifiers	Range of Values Represented	Number of Bytes
INTEGER*1	0 to 255	1
INTEGER*2	-32768 to 32767 (-2^{15} to $2^{15} - 1$)	2
INTEGER or INTEGER*4	-2147483648 to 2147483647 (-2^{31} to $2^{31} - 1$)	4
INTEGER*8	-9223372036854775808 to 9223372036854775807 (-2^{63} to $2^{63} - 1$)	8

For each of the integer data types, except INTEGER*1, negative values are represented internally in two's complement form.

Because Gould CSD hardware does not contain double precision integer arithmetic instructions such as multiply and divide, all such operations must be implemented by software. Under this implementation, division requires the use of a dividend's complement. Since a maximum negative INTEGER*8 value does not have a complement, division with this value as the dividend will result in an incorrect quotient.

3.2.1 Integer Constants

An integer constant represents an exact value of the integer data type. The form of an integer constant is a sign (optional for positive values) followed by a string of digits as shown below.

$$[\pm]\ d_1 d_2 \ldots d_i$$

d_i Digits interpreted as a decimal (base 10) number.

Rules for Use

- Integers must contain from 1 to 19 digits, not including leading zeros.
- A preceding plus sign (+) is optional for positive numbers. A minus sign (-) is required for negative numbers.
- No decimal point (.) or comma (,) is allowed.
- Blanks are allowed, but are not significant.
- The amount of precision indicated by the constant will determine the type of integer storage required.

Examples

Valid Integer Constants

```
0
45
205
-94761
```

Invalid Integer Constants	Error Analysis
38.	Contains a decimal point
99282939202992829392056483 6	Exceeds the allowable range
3,947	Contains an embedded comma

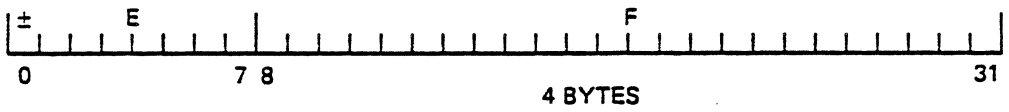
3.3 Real Data Type

The real data type represents an approximation of a real number over a larger range than allowed in the integer data types. The values can be positive, negative, or zero.

How Specified in Type Specifiers	Range of Values Represented	Number of Bytes
REAL or REAL*4	16^{-65} to 16^{63}	4

The range of real numbers represented by the real data type is approximately $5*10^{-79}$ to $7*10^{75}$ (in general, 6 to 7 digits of decimal accuracy).

The internal representation of the real data type in storage is as follows:



830623-1B

A real number can be represented by a sign, a fraction, and an exponent. The sign (bit 0) applies to the fraction and denotes a positive or negative value. If the sign is negative, the entire word is the two's-complement of the absolute value. The fraction is a hexadecimal normalized number with the radix point to the left of the high order fraction bit (bit 8). The exponent (bits one through seven) is a seven-bit number from which 64 (decimal) is subtracted to obtain the power to which base 16 is raised.

All operands are assumed normalized before being operated upon. A positive number is normalized when the value of the fraction is less than one and greater than or equal to one sixteenth. A negative number is normalized when its two's complement is normalized. The result of an operation with normalized operands will be properly normalized. Nonnormalized operands may produce incorrect results.

3.3.1 Real Constants

A real constant represents an approximate value of a real number with the precision and range of the real data type. The form of a real constant is an optional sign (for positive values), an integer part, a decimal point, and a fractional part, in that order. Both the integer part and the fractional part are strings of digits; either of these parts can be omitted, but not both. A real constant can contain an exponent, which is formed by the letter E followed by an optionally signed integer constant. Therefore, a real constant can be expressed in one of the following forms:

$\pm r$ or $\pm iE\pm e$ or $\pm rE\pm e$

where r is one of the following forms:

i.
.f
i.f

and i, f, and e are each sequences of digits representing integer, fraction, and exponent, respectively.

The interpretation of the $E\pm e$ part of the real constant is a power of ten exponent, so that $\pm rE\pm e$ is interpreted as the value of the real number times $10^{\pm e}$.

Rules for Use

- A plus sign (+) is optional.
- If the constant preceding $E\pm e$ contains more significant digits than the precision for real data allows, truncation occurs and only the most significant digits in the range are processed.

Examples

Valid Real Constants	Exponent Interpretation	Decimal Value
+0.	N/A	0.0
-493.6843	N/A	-493.6843
4.0E+0	4.0×10^{0}	4.0
24938.36E+1	24938.36×10^{1}	249383.6
8E2	8.0×10^{2}	800.0
9.0E3	9.0×10^{3}	9000.0
9.0E+03	9.0×10^{3}	9000.0
6E-03	6.0×10^{-3}	0.006

Invalid Real Constants	Error Analysis
4	Missing a decimal point or a decimal exponent
4,580.3	Embedded comma
3.4E+140	Magnitude outside the allowable range; that is, $3.4 \times 10^{140} > 16^{63}$
45.7E+97	Magnitude outside the allowable range; that is, $45.7 \times 10^{97} > 16^{63}$
68.3E-95	Magnitude outside the allowable range; that is, $68.3 \times 10^{-95} < 16^{-65}$

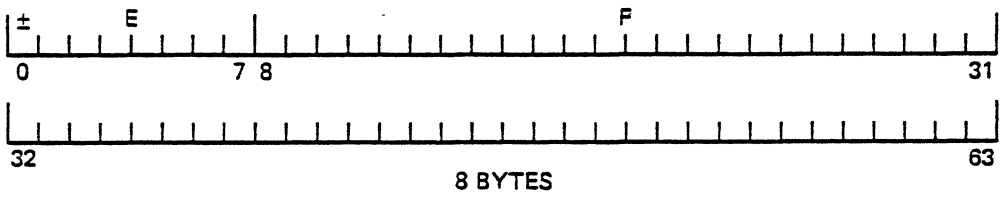
3.4 Double Precision Data Type

The double precision data type represents an approximation of a real number over a larger range than allowed in integer data types. A data item with the double precision data type can have a positive, negative, or zero value. The only difference between double precision and real data types is in the precision of the approximation and number of bytes of storage required.

How Specified in Type Specifiers	Range of Values Represented	Number of Bytes
DOUBLE PRECISION or REAL*8	16^{-65} to 16^{63}	8

The range of real numbers represented by the double precision data types is approximately $5*10^{-79}$ to $7*10^{75}$ (in general, 15 to 16 digits of accuracy).

The internal representation of the double precision data type is as follows:



830623-3B

The exponent and fraction are defined in the same manner as the real data type. The increased fraction size provides more precision within the same range of magnitude.

3.4.1 Double Precision Constants

A double precision constant represents an approximate value of a real number with the precision and range of the double precision data type. A double precision constant is formed in the same manner as a real constant. The form and interpretation of a double precision exponent are identical to those of the real exponent, except that the letter D is used instead of the letter E.

The value of a double precision constant is the product of the constant that precedes the D and the power of ten indicated by the integer following the D.

Examples

Valid Constants for Double Precision	Exponent Interpretation	Decimal Value
-5.9D03	-5.9×10^3	-5900.0
or		
.59D03	$.59 \times 10^3$	590.0
5.9D03	5.9×10^3	5900.0
5.9D+03	5.9×10^3	5900.0
5.9D+3	5.9×10^3	5900.0
5.9D0	5.9×10^0	5.9
8D03	8.0×10^3	8000.0

Invalid Constants for Double Precision	Error Analysis
4	Missing a decimal point or a decimal exponent
4,580.3	Embedded comma
4.D	Missing the one- or two-digit integer constant following the D. Note that it is not interpreted as 4.0×10^0
3.4D+140	Magnitude outside the allowable range; that is, $3.4 \times 10^{140} > 16^{63}$
45.7D+97	Magnitude outside the allowable range; that is, $45.7 \times 10^{97} > 16^{63}$
68.3D-95	Magnitude outside the allowable range; that is, $68.3 \times 10^{-95} < 16^{-65}$
68.3	Missing D exponent

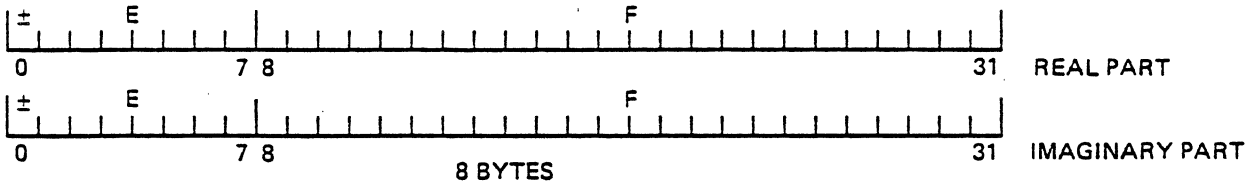
3.5 Complex Data Types

Complex data types represent an approximation of complex numbers in the form of an ordered pair of real or double precision data types. The first of the pair approximates the real part, and the second approximates the imaginary part. The two complex data types differ only in the precision of the approximation and number of bytes of storage required.

How Specified in Type Specifiers	Range of Values Represented	Number of Bytes
COMPLEX or COMPLEX*8	16^{-65} to 16^{63} (6 to 7 decimal digits)	8
COMPLEX*16	16^{-65} to 16^{63} (15 to 16 decimal digits)	16

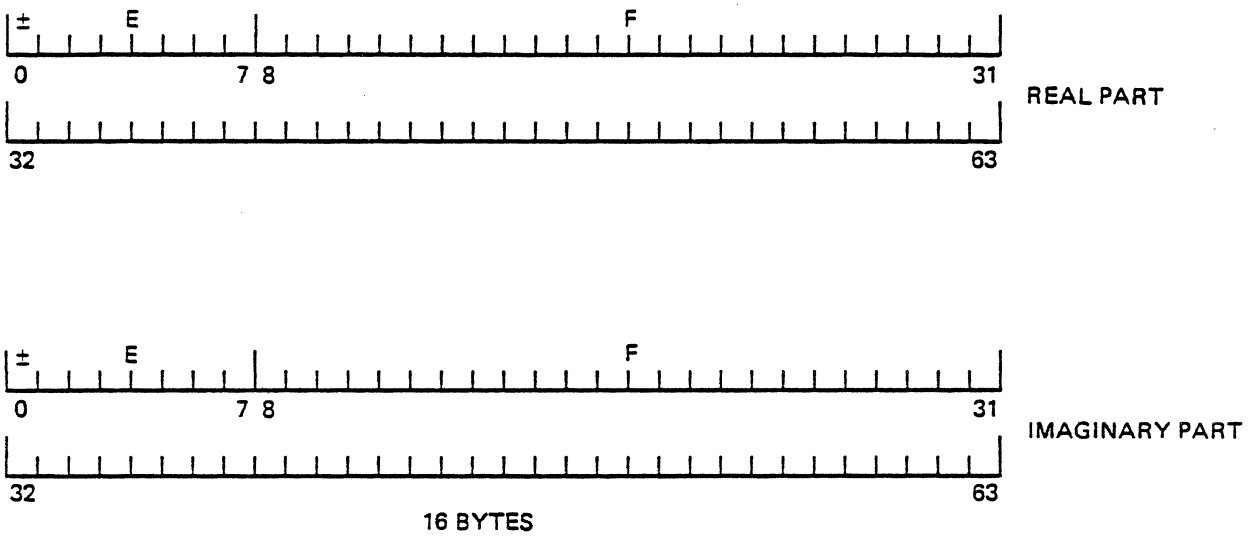
The range of each half (real/imaginary) of each complex data type is the same as the corresponding real or double precision data type.

The internal representation of COMPLEX*8 is as follows:



830623-4B

The internal representation of the COMPLEX*16 data type is as follows:



830623-2B

3.5.1 Complex Constants

A complex constant represents an approximation of a complex number. The form of a complex constant is a left parenthesis followed by an ordered pair of real, double precision, or integer constants separated by a comma, and followed by a right parenthesis. The components of the ordered pair need not be of the same type.

Automatic type conversion is made to COMPLEX*8 unless one of the constants is a double precision constant, in which case the conversion is to COMPLEX*16. Thus, (1,2) is legal and is identical to (1.0,2.0). (1E0,2D0) is legal and is identical to (1.0D0,2.0D0).

The first constant of the pair is the real part of the complex constant and the second is the imaginary part.

(real,imaginary)

Rules for Use: Complex Word Constants

- A pair of real or integer constants represents the real and imaginary parts of a complex word constant.
- The rules for integer and real constants govern the construction of each of the two parts.

Rules for Use: Complex Doubleword Constants

- A pair of integer, real or double precision constants represents the real and imaginary parts of a complex doubleword constant. One of the constants must be double precision.
- The rules for integer, real and double precision constants govern the construction of each of the two parts.

Examples

Valid Complex Constants	Value ($i = \sqrt{-1}$)
(4.3,-9.54)	4.3 -9.54i
(-3.0E+03,.12E+02)	-3000.+ 12.0i
(3.6D+2,1.6734839)	360.+ 1.6734839i
(36D+2,29D+3)	3600.+ 29000.i

Invalid Complex Constants	Error Analysis
(36D+2;29D+3)	Constants must be separated by a comma

3.6 Bit Data Type

The bit data type represents the logical values true and false.

How Specified in Type Specifiers	Values Represented	Number of Bytes
BIT	true and false	1/8

3.7 Logical Data Types

Logical data types are used to represent the logical values true and false.

How Specified in Type Specifiers	Values Represented	Number of Bytes
LOGICAL*1	true and false	1
LOGICAL or LOGICAL*4	true and false	4

3.7.1 Logical or Bit Constants

Logical or bit constants represent the value true or false for use with logical or bit data types. Therefore, there are only two forms a logical constant can have:

.TRUE. logical value true
.FALSE. logical value false

3.8 Character Data Type

The character data type represents strings of characters. The character string can consist of from 1 to 4095 ASCII characters. The blank character is valid and significant in a character datum. The length of a character datum is the number of characters in the string, and it is specified by the character type statement (see Chapter 7). One character requires one byte of storage.

Each character in a string has a character position that is numbered consecutively 1, 2, 3, etc.. The number indicates the sequential position of the character in the string, from left to right.

3.8.1 Character Constant

A character constant represents a character string value with the character data type. The form of a character constant is an apostrophe followed by a nonempty string of characters followed by an apostrophe. The delimiting apostrophes are not part of the datum represented by the constant. An apostrophe within the string is represented by two consecutive apostrophes with no intervening characters. In a character constant, blanks embedded between the delimiting apostrophes are significant.

The length of a character constant is the number of characters between the delimiting apostrophes, except that each pair of consecutive apostrophes counts as a single character; the delimiting apostrophes are not counted.

Rules for Use

- All characters, including blanks, are significant.
- An apostrophe within the constant is represented by two consecutive apostrophes.
- The length of a character constant must be greater than zero.
- The quotation mark is used as an escape. The character following the quotation mark is included in the string but the quotation mark is not included.

Examples

Valid Character Constants	Value
'STOP'	STOP
'FORMULA'	FORMULA
'CAN' 'T'	CAN'T
''''	'
'"'	"

Invalid Character Constants	Error Analysis
FORMULA	Apostrophes omitted
'CAN'T'	Two aprostrophes needed

3.8.2 Character Constants versus Hollerith Constants

Associated with each character constant are two values: a start address and a length. This is in contrast with a Hollerith constant, which consists of only the byte, halfword, word, or doubleword containing the constant. Quoted strings can be assigned to variables because the context of the quoted string (character or Hollerith) is determined by the type of variable the string is being assigned to. However, when quoted strings are used as arguments to subroutines or I/O lists, the compiler generates, by default, a character constant. If you want these quoted string arguments to be built as Hollerith constants, option 10 must be set at compile time.

Note that whenever a quoted string constant is a parameter in the calling list of an M:xxxxxx subroutine or function call, option 10 must be set.

3.9 Hollerith Constants

A Hollerith constant is a string of characters, delimited as follows:

- The string can be preceded by nH, where n is the number of characters in the string. This form is left-justified and blank-filled.
- The string can be preceded by nR, where n is the number of characters in the string; R indicates that it is right-justified and zero-filled. Note that n cannot exceed 8 for right-justified Hollerith constants.

The form of a Hollerith constant is a nonzero, unsigned, integer constant n followed by the letter H or R, followed by a string of exactly n contiguous characters. The string can consist of any characters capable of representation in the processor. The string of n characters is the Hollerith datum.

nHs

nRs

n	An unsigned integer specifying the number of characters in the string.
s	A string of characters.

Hollerith data can be associated with various data types by using assignment, data initialization, and CALL statements and through input/output operations using the Aw format specifications. Hollerith data can be used in CALL statements or function reference argument lists as data initialization values (including assignment statements). Hollerith data is stored as follows:

Data Type	Number of Characters
INTEGER*8	8
DOUBLE PRECISION (REAL*8)	8
REAL (REAL*4)	4
INTEGER (INTEGER*4)	4
INTEGER*2	2
INTEGER*1	1

Rules for Use

- All characters, including blanks, are significant in Hollerith character strings.
- If nRs is used, the n must be less than or equal to eight.

Examples

Valid Hollerith Constants	Represents
1HA	A
2RGT	GT
5HGO∅ON	GO∅ON

Invalid Hollerith Constants	Error Analysis
5HUNABLE	Too long

3.10 Hexadecimal Constants

A hexadecimal constant is a hexadecimal number (base 16) formed from the set 0 through 9 and A through F. It is coded in one of the following forms:

$X'd_1 \ldots d_i'$ or $Z'd_1 \ldots d_i'$ or nZs

n — An unsigned integer specifying the number of digits in the string. The range of n is between 1 and 16. Constants are typed as follows:

1,n,8	INTEGER (INTEGER*4)
9,n,16	INTEGER (INTEGER*8)

s — A string of hexadecimal digits.

d_i — 0 through 9 and A through F.

Note that a hexadecimal constant cannot contain embedded blanks.

Examples

Valid Hexadecimal Constants	Decimal Value
Z'098700FD'	159842557
Z'FFFFE560'	-6816
8Z0000FFFF	65535
2Z0F	15
X'64'	100

Invalid Hexadecimal Constants	Error Analysis
21Z347	n must be $\leq$ 16
Z527	No value for n
2Zb0F	Contains an embedded blank

3.11 Binary Constants

Binary constants represent a number in binary notation (base 2) and have the following form:

$B'd_1 \ldots d_i'$

d_i 0 or 1. Embedded blanks are not permitted.

Examples

Valid Binary Constants	Decimal Value
B'101'	5
B'1101'	13

Invalid Binary Constants	Error Analysis
B'1001	Missing apostrophe
B'1012'	d must be 0 or 1
B'11b1'	Embedded blank

3.12 Octal Constants

Octal constants represent a number in octal notation (base 8) and have the following form:

$O'd_1 \ldots d_i'$

d_i 0 through 7. Embedded blanks are not permitted.

Examples

Valid Octal Constants	Decimal Value
O'10'	8
O'77'	63

Invalid Octal Constants	Error Analysis
O'68'	d must not be > 7
O'7b7'	Embedded blank

3.13 Symbolic Names

A symbolic name identifies an element in a program unit. Symbolic names can consist of any combination of from one to eight alphanumeric characters (more than eight characters may be specified but only the first eight are significant). The first character of a symbolic name must be an alphabetic character; blank characters are ignored.

The colon (:) and underline (_) characters can be used within symbolic names; however, the use of the colon should be avoided in a context where it would be a legal delimiter (e.g., character substrings, variable lower dimension bounds).

3.13.1 Implicit Typing Conventions

The following implicit typing conventions are assumed unless a symbolic name has been explicitly declared to be a particular data type, an IMPLICIT statement has been used to change the convention, or the IMPLICIT NONE statement has been used.

- Symbolic names beginning with I, J, K, L, M, or N assume the INTEGER (INTEGER*4) data type.
- Symbolic names beginning with letters other than I, J, K, L, M, or N assume the REAL (REAL*4) data type.

Examples

Valid Symbolic Names

```
B8302
GAMMA
HO_ME
M:WAIT
```

Invalid Symbolic Names	Error Analysis
#GAMMA	Symbol used as first character
7XX	Number used as first character

Spellings generally associated with specific statements can also be symbolic names. The example below illustrates statements containing symbolic names that are the same as FORTRAN 77+ keywords.

```
GOTO3 = 1.0
DO5I =1.7
```

GO TO and DO are FORTRAN 77+ keywords; however, in these examples, GOTO3 and DO5I are both symbolic names. Note the similarity between the statements DO5I=1.7 and DO 5 I=1,7. The compiler recognizes the first as an assignment statement rather than as a DO statement. Use the IMPLICIT NONE statement to avoid confusing one kind of statement with another (refer to Chapter 7).

Symbolic names are used to identify the following items of a FORTRAN 77+ source program:

- Variables
- Constants
- Arrays
- Array elements
- Functions
- Subroutines
- External procedures
- Common block names
- NAMELIST names
- Block data subprograms
- Programs

3.14 Variables

A variable is a symbolic name that identifies a storage location. Variables are data whose values can change during program execution.

Every variable must be assigned a value before it is referenced or the value of the variable will be undefined. An initial value can be assigned in a DATA statement, or it can be defined during program execution.

Variables can be any of the following data types: INTEGER*1, INTEGER*2, INTEGER (INTEGER*4), INTEGER*8, REAL (REAL*4), DOUBLE PRECISION (REAL*8), COMPLEX*8, COMPLEX*16, BIT, LOGICAL*1, LOGICAL (LOGICAL*4), or CHARACTER.

3.15 Arrays

An array is an ordered set of data characterized by the property of dimension and identified by a single symbolic name. An array name must conform to the rules for writing symbolic names. Individual storage locations, called array elements, are referenced by subscripts appended to the array name.

The following FORTRAN 77+ statements can establish arrays:

- Type declaration
- DIMENSION
- COMMON
- EXTENDED BLOCK

These statements can contain array declarations that define the name of an array, the number of dimensions in an array, and the number of elements in each dimension.

An array can have from 1 to 7 dimensions. To reference a specific value in an array, specify the subscript values of each dimension for the particular element (see Figure 3-1).

The use of an array name in an assignment statement without subscripts enables you to assign a single value to every element of the array.

3.15.1 Array Declarators

An array declarator is used in specification statements to specify the symbolic name that identifies an array within a program unit and to indicate the properties of that array.

An array declarator has the form:

a (d [,d] . . .)

a The symbolic name of the array.

d A dimension declarator; it can specify both a lower bound and upper bound as follows:

[dl:] du

dl the lower bound of the dimension.

du the upper bound of the dimension.

- Within a program unit, an array name can be in one array declarator only.
- If dl is a variable name that contains a colon (:), the name must be in parentheses. Likewise, if dl is an expression containing a variable name with a colon, the expression must be in parentheses.

3.15.2 Dimensions of an Array

The number of dimension declarators indicates the number of dimensions in the array and can range from one to seven.

The value of the lower bound dimension declarator can be negative, zero or positive; the upper bound dimension declarator must be greater than or equal to the lower bound dimension declarator. The number of elements in a dimension is du-dl+1. If the lower bound is not specified, it is assumed to be one, and the value of the upper bound specifies the number of elements in that dimension.

Examples

DIMENSION A(2:10) Specifies an array A; the dimension declarator 2:10 indicates a lower bound of 2 and an upper bound of 10; the number of elements in the dimension is 9.

DIMENSION A(10) Specifies an array A; the single dimension declarator indicates the dimension contains 10 elements.

Lower and upper dimension bounds are integer arithmetic expressions. Note the following:

- For dummy arrays within subprograms, the upper dimension bound of the last dimension in an array can be an asterisk. For example, the first DIMENSION example above could be written:

```
DIMENSION A(2:*)
```

- For dummy arrays within subprograms, each operand of a dimension-bound expression can be an integer constant, an integer dummy argument, or an integer variable in a common or extended block. The last dimension-bound expression can be an asterisk.

- A dimension-bound expression must not contain a function or an array element reference.

- Integer variables can be in dimension-bound expressions only in dummy array declarators.

- If the type of a symbolic name of a constant or variable in a dimension-bound expression is not the same as the name's default implicit type (see Section 3.13.1), it must be specified as INTEGER (INTEGER*4) by an IMPLICIT statement or a type statement before the name is used in a dimension-bound expression.

3.15.3 Subscripts

A subscript is a parenthesized list of expressions appended to the name of the array it qualifies. A subscript determines which element in an array is being referenced. A subscript expression is an integer expression that can contain array element references and function references.

An array element is considered undefined until it is assigned a value. If reference is made to an undefined array element, the value of the element will be unknown and unpredictable.

The form of a subscript is

$(s_1 [,\ldots,s_7])$

s a subscript expression.

The following rules govern the use of subscripts:

- A subscript contains one to seven subscript expressions enclosed in parentheses.

```
D 1 (1)
  .
  .
  .
D 7 (1,2,3,4,5,6,7)
```

- Two or more subscript expressions within the parentheses must be separated by commas.

```
D(I,2,K)
X(2*J-3,7)
```

- The number of subscript expressions must be the same as the specified dimensions of the array declarator.

- A subscript expression can be of any arithmetic type. The value of the expression, however, is converted to an integer value as necessary.

- Subscripts themselves can be subscripted.

```
C(N(I), J-2)
```

3.15.4 Array Storage

In FORTRAN 77+ an array is stored as a linear sequence of values. A one-dimensional array is stored with its first element in the first storage location and its last element in the last storage location of the sequence. A multidimensional array is stored in row major order; that is, the leftmost subscript, which represents the row, varies most rapidly. Figure 3-1 illustrates array storage in one, two, and three dimensions.

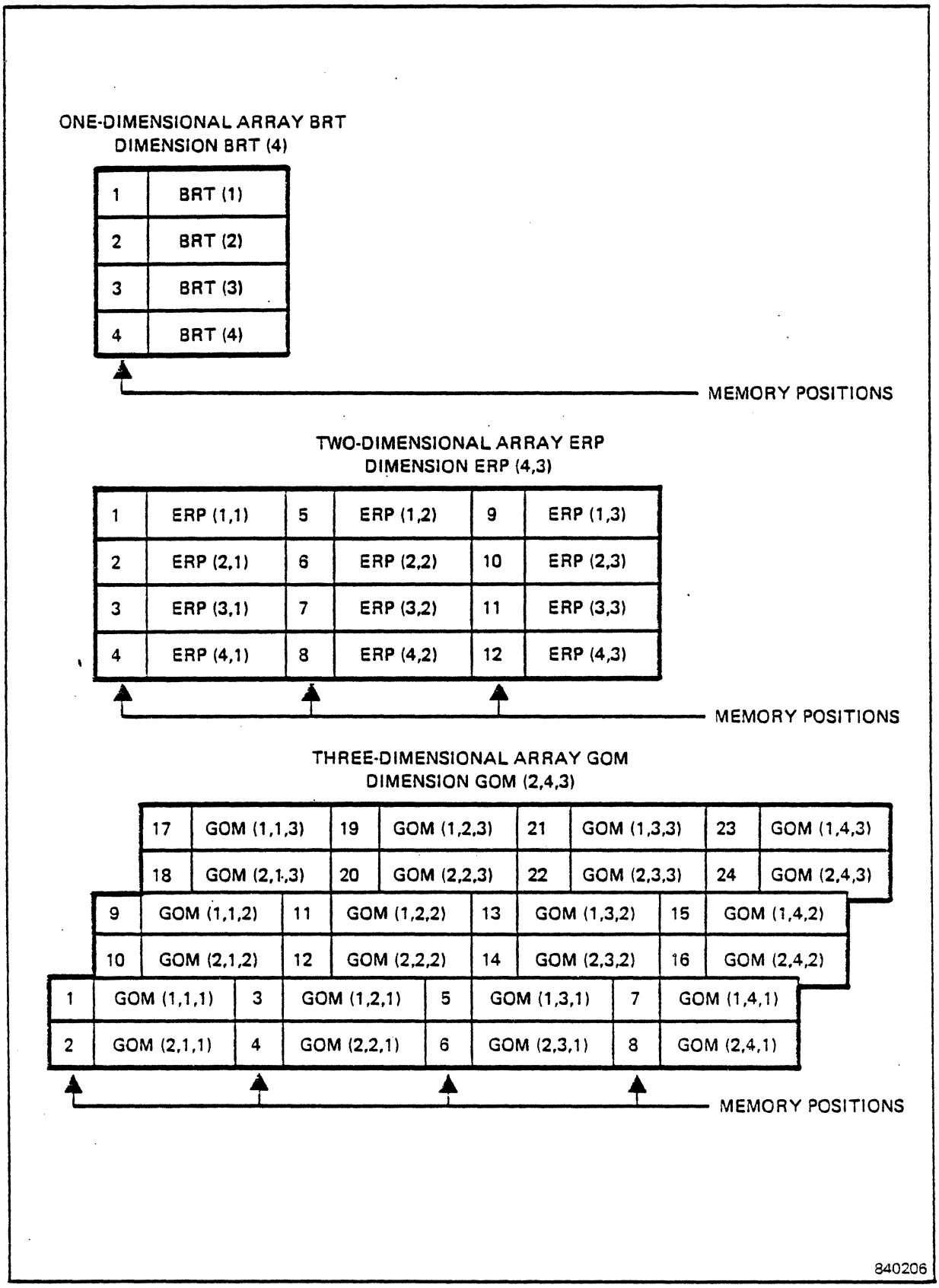


Figure 3-1. Array Storage

CHAPTER 4

EXPRESSIONS

4.1 General

An expression consists of a single operand or a string of operands connected by operators. Operands can be constants, variables, array elements, or function references. Operators can be unary (operating on a single operand) or binary (operating on a pair of operands). An expression can contain subexpressions (expressions enclosed in parentheses) as operands. There are four types of expressions: arithmetic, character, relational, and logical.

4.2 Arithmetic Expressions

Arithmetic expressions express numeric computations; evaluation of an arithmetic expression produces a numeric value.

Arithmetic expressions consist of constants, variables, array elements, function references, and subexpressions of any data type (except BIT, LOGICAL*1, LOGICAL (LOGICAL*4), or CHARACTER).

The operators and their meanings are:

+	Addition
-	Subtraction or negation
*	Multiplication
/	Division
**	Exponentiation

Rules for Use

- Contiguous arithmetic operators are not allowed. For example,

  X*-Y

  is prohibited. However,

  X*(-Y)

  is allowed because the minus sign designates a negative value rather than the subtraction operator.

- Parentheses, when included to form subexpressions, can be used freely to determine the order of evaluation in mathematical expressions. Parentheses do not indicate multiplication. The only acceptable indication of multiplication is the asterisk (*) appearing between multiplier and multiplicand.

CD	Unacceptable to mean C times D
C(D)	Unacceptable to mean C times D
C*D	Acceptable to mean C times D

- A variable, array element, constant, or function reference standing alone is an expression.

S	217
A(I)	17.26
JOBNO	SQRT(2.0)

- If E is an expression whose first character is not an operator, then +E and -E are expressions.

-S	+217
+A(I)	+17.26
+JOBNO	-SQRT(X)
-217	

- If E is an expression, (E) is a subexpression meaning the quantity E is taken as a unit.

(-S)	(217)
(A(I))	(17.26)
(JOBNO)	(-SQRT(X))

- If E is an expression whose first character is not an operator and F is any expression, then F+E, F-E, F*E, F/E, and F**E are all expressions.

  ```
  -(B(I,JO)+SQRT(X))

  1.7**(A+5.0)
  ```

- Subexpressions can be nested, as indicated below:

  ```
  A*(Z-((Y+M)/T))**J
  ```

  where (Y+M) is the innermost subexpression, ((Y+M)/T) is the next innermost, and (Z-((Y+M)/T)) the next. In all expressions, the number of left parentheses must equal the number of right parentheses.

- The order of precedence determines the data type of a value resulting from any valid binary operator.

Type	Precedence
COMPLEX*16	1 (highest)
COMPLEX (COMPLEX*8)	2
DOUBLE PRECISION (REAL*8)	3
REAL (REAL*4)	4
INTEGER*8	5
INTEGER (INTEGER*4)	6
INTEGER*2	7
INTEGER*1	8 (lowest)

- An operand of one arithmetic data type can be combined with the operand of any other arithmetic data type by use of arithmetic operators (+, -, *, /, **). The type of the result is the same as that of the operand having the higher precedence. The operand of lower precedence is converted to the type of high precedence before the operation is performed. Table 4-1 indicates the resulting data type of all arithmetical combinations by type.

Table 4-1 (Page 1 of 2)
Expression Type Determination from Arithmetic
Operands of Different Types

	Operator	Type of Operand Two			
	+, -, *, /, **	INTEGER*1	INTEGER*2	INTEGER (INTEGER*4)	INTEGER*8
Type of Operand One	INTEGER*1	INTEGER*1	INTEGER*2	INTEGER (INTEGER*4)	INTEGER*8
	INTEGER*2	INTEGER*2	INTEGER*2	INTEGER (INTEGER*4)	INTEGER*8
	INTEGER (INTEGER*4)	INTEGER (INTEGER*4)	INTEGER (INTEGER*4)	INTEGER (INTEGER*4)	INTEGER*8
	INTEGER*8	INTEGER*8	INTEGER*8	INTEGER*8	INTEGER*8
	REAL	REAL	REAL	REAL	REAL
	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION
	COMPLEX*8	COMPLEX*8	COMPLEX*8	COMPLEX*8	COMPLEX*8
	COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16

Table 4-1 (Page 2 of 2)
Expression Type Determination from Arithmetic Operands of Different Types

Type of Operand One \ Operator: +, -, *, /, **	Type of Operand Two: REAL (REAL*4)	DOUBLE PRECISION (REAL*8)	COMPLEX (COMPLEX*8)	COMPLEX*16
INTEGER*1	REAL	DOUBLE PRECISION	COMPLEX*8	COMPLEX*16
INTEGER*2	REAL	DOUBLE PRECISION	COMPLEX *8	COMPLEX*16
INTEGER (INTEGER*4)	REAL	DOUBLE PRECISION	COMPLEX*8	COMPLEX*16
INTEGER*8	REAL	DOUBLE PRECISION	COMPLEX*8	COMPLEX*16
REAL	REAL	DOUBLE PRECISION	COMPLEX*8	COMPLEX*16
DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	COMPLEX*8	COMPLEX*16
COMPLEX*8	COMPLEX*8	COMPLEX*8	COMPLEX*8	COMPLEX*16
COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16

4.2.1 Evaluation of Arithmetic Expressions

Expressions are generally evaluated according to the following rules:

- Subexpressions are generally evaluated first. If there are nested subexpressions, the innermost is evaluated, then the next innermost, and so on, until all subexpressions have been evaluated.
- Within subexpressions, or wherever parentheses do not govern the order of evaluation, the hierarchy of operations in the order of precedence is

  Function evaluation

  Subscript evaluation

  Exponentiation

  Multiplication and division

  Addition and subtraction

For example,

```
F+((A+B)*C)+D**2
```

is evaluated in the following sequence:

```
E1=A + B
E2=E1*C
E3=D**2
E4=F + E2
E5=E4 + E3
```

The expression

```
A*(Z-((Y+M)/T))**J+VAL
```

is evaluated in the following sequence:

```
E1=Y+M
E2=E1/T
E3=Z-E2
E4=E3**J
E5=A*E4
E6=E5+VAL
```

- Whenever operations of the same level of hierarchy are involved, evaluation proceeds generally from left to right, except as indicated by the interpretation of the last expression below:

Expression	Interpretation
W*X/Y*Z	((W*X)/Y)*Z
B**2-4.*A*C	(B**2)-((4.*A)*C)
X/Y/Z	(X/Y)/Z
-X**3	-(X**3)
X**Y**Z	X**(Y**Z)

For the commutative operators + and *, evaluation of operations at the same hierarchy normally proceeds from left to right; however, the order of evaluation can vary when values of terms or factors are present in registers because of a prior evaluation.

For example,

```
X-Y-Z
```

can be interpreted as

```
(-Z-Y)+X
```

- Use of an array element name requires the evaluation of its subscript. The same rules that govern the evaluation of expressions apply to subscripts; however, the value of each subscript expression is converted to integer (if necessary) before any address calculations are performed.

- If a function reference causes definition of an actual argument to the function, that argument or any associated entities must not appear elsewhere in the same statement. For example:

```
A(I) = F(I)
WRITE(6,50) I, F(I)
Y = F(I) + I
```

are prohibited if the reference to F defines I.

4.3 Character Expressions

A character expression, which is used to create a character string, is composed of a sequence of one or more character primaries (see below) separated by the concatenation operator. Evaluation of a character expression produces a result of type CHARACTER.

The character primaries are

- Character constant
- Symbolic name of a character constant

- Character variable reference
- Character array element reference
- Character substring reference
- Character function reference
- Character expression enclosed in parentheses

Combining one or more character operands with the concatenation operator (//) and parentheses forms a more complicated character expression.

The forms of a character expression are:

- Character primary
- Character expression // character primary

4.3.1 The Value of a Character Expression

The result of a concatenation operation is a character string formed by successive left-to-right linking of the character expression elements (primaries). The length of a character expression is the sum of the character primary lengths.

For example, the value of

```
'AB' // 'CDE'
```

is the string ABCDE, which has a length of five.

4.3.2 Character Constant Expression

A character constant expression is a character expression in which each primary is a character constant, the symbolic name of a character constant, or a character constant expression enclosed in parentheses.

4.3.2.1 Character Substring

A character substring is a contiguous portion of a character string and is of type character. A character substring is identified by a substring name and can be assigned values and referenced.

There are two ways to identify a substring name:

v([e_1]:[e_2])

and

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

v — A character variable name.

a(s[,s]...) — A character array element name.

e_1 An integer expression called a substring expression. The value e_1 specifies the leftmost character position of the substring. The expression e_1 must not contain any symbolic names having a colon (:) in the name at an unparenthesized level.

e_2 An integer expression called a substring expression. The value e_2 specifies the rightmost character position.

The values of e_1 and e_2 must be such that

$$1 \leq e_1 \leq e_2 \leq \text{len}$$

len The length of the character variable or array element.

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

The default value for e_1 is one, and the default value for e_2 is the value of len. Both e_1 and e_2 can be omitted, in which case the following apply:

v(:) is equivalent to v

and

a(s[,s])(:) is equivalent to a(s[,s])

The length of a character substring is e_2-e_1+1.

A substring expression (e_1 or e_2 above) can contain array element references and function references. Evaluation of a function must not alter the value of any other expression within the substring name.

Rules for Use

- A character expression and its operands must identify values of type CHARACTER.
- A character expression must not involve concatenation of a dummy argument whose length specification is an asterisk in parentheses except in a character assignment statement.
- No part of a character string that is being assigned a value by an assignment statement can be a part of the expression on the right side of the equal sign.

4.4 Relational Expressions

A relational expression consists of two arithmetic expressions or two character expressions separated by a relational operator.

4.4.1 Relational Operators

Relational operators test for a relationship between two arithmetic expressions or between two character expressions. These operators are:

Relational Operator	Definition
.GT.	Greater than (>)
.GE.	Greater than or equal to (≥)
.LT.	Less than (<)
.LE.	Less than or equal to (≤)
.EQ.	Equal to (=)
.NE.	Not equal to (≠)

Delimiting periods are a required part of each operator.

4.4.2 Arithmetic Relational Expressions

Relational operators express an arithmetic condition that can be either .TRUE. or .FALSE. . Arithmetic relational operators can combine only arithmetic expressions whose types are INTEGER, REAL, or DOUBLE PRECISION; however, COMPLEX types can use the relational operators .NE. and .EQ..

The following examples illustrate valid and invalid relational expressions, given the valid variable names:

Variable Name	Type
ALPHA,B	real
I, K, M	integer
L	logical
C	complex
X	character

Valid Relational Expressions

```
I .LT. K
B**3.6 .EQ. (4*ALPHA+9)
5 .GT. 1.2*ALPHA
8 .NE. 29.1D+03
```

Invalid Relational Expressions	Error Analysis
C .LE. (1.2,4.3E2)	Complex quantities can only appear in relational expressions using the .EQ. or .NE. operators.
L .EQ. .TRUE.	Logical quantities can never be joined by relational operators.
E**3 .LT 38.E7	Missing period, which is part of the relational operator.
.NE. 58	Missing arithmetic expression before the relational operator.
X .EQ. 5	Relational operators cannot join a character expression and an arithmetic expression.

Examples of valid relational expressions are presented below; values for previously defined variables are:

```
ALPHA = 1.2
B     = 3.1
I     = 2
K     = 5
M     = 7
```

Expression	Result
I .LT. K	.TRUE.
B**3.6 .EQ. (4*ALPHA+9)	.FALSE.
5 .GT. 1.2*ALPHA	.TRUE.
B .NE. 29.1D+03	.TRUE.

4.4.3 Character Relational Expressions

A character relational expression is interpreted as the logical value .TRUE. if the values of the operands satisfy the relation specified by the operator; otherwise, it is interpreted as the logical value .FALSE..

For character relational expressions, .LT. means precedes in the ASCII collating sequence, and .GT. means follows in the ASCII collating sequence. Relational operators can also be used with character expressions. If the operands are of unequal length, the shorter operand is considered to be extended on the right with blanks to the length of the longer operand. A blank precedes letters and digits in the collating sequence. The collating sequence does not influence the result of operators .EQ. and .NE..

Character relational expressions are evaluated at run-time and can not be used in constructing constants at compile-time. Therefore, statements such as

```
PARAMETER (LOGIC1='ABC' .NE. 'DEF')
```

are not allowed.

The following are examples of legal character relational expressions and their results:

Expression	Result
'ALFA' .LT. 'ZETA'	.TRUE.
'SOME' .EQ. 'ANY'	.FALSE.
'ABC' .GT. 'XYZ'	.FALSE.
' 9' .LT. '8 '	.TRUE.

4.5 Logical Expressions

A logical expression can be a single logical operand or a combination of logical operands and logical operators.

The value of a logical expression is always either .TRUE. or .FALSE.. A logical expression can be either simple or compound. A simple logical expression can be any of the following:

- A logical constant
- A logical variable
- A logical array element
- A relational expression
- A logical expression enclosed in parentheses
- A logical function reference

A logical expression may have one of the following forms:

$o_1 e_1$ or $e_1 o_2 e_2$

e_1 A logical operand.

o_1 A unary logical operator.

o_2 A binary logical operator.

e_2 A logical operand.

4.5.1 Logical Operators

For the following examples, let T be a .TRUE. expression and F be a .FALSE. expression.

Operator	Example	Meaning
Unary		
.NOT.	.NOT. E	This expression has the value .TRUE. only if E is .FALSE.; it has the value .FALSE. only if E is .TRUE..

Expression	Result
.NOT. T	.FALSE.
.NOT. F	.TRUE.

Binary

.AND. E .AND. G

This expression has the value .TRUE. only if E and G are both .TRUE.; it has the value .FALSE. if either E or G is .FALSE..

Expression	Result
T .AND. T	.TRUE.
T .AND. F	.FALSE.
F .AND. T	.FALSE.
F .AND. F	.FALSE.

.OR. E .OR. G

This expression has the value .TRUE. if either E or G is .TRUE.; it is .FALSE. only if both E and G are .FALSE.. This is known as the inclusive OR.

Expression	Result
T .OR. T	.TRUE.
T .OR. F	.TRUE.
F .OR. T	.TRUE.
F .OR. F	.FALSE.

.EQV. E .EQV. G

This expression has the value .TRUE. only if E and G are both .TRUE., or if E and G are both .FALSE.. It has the value .FALSE. only if the values of E and G are different.

Expression	Result
T .EQV. T	.TRUE.
T .EQV. F	.FALSE.
F .EQV. T	.FALSE.
F .EQV. F	.TRUE.

.NEQV. E .NEQV. G

This expression has the value .TRUE. only if E and G are different; it is .FALSE. if the values of E and G are the same.

Expression	Result
T .NEQV. T	.FALSE.
T .NEQV. F	.TRUE.
F .NEQV. T	.TRUE.
F .NEQV. F	.FALSE.

.EOR.

.XOR.

The logical operators .EOR. and .XOR. are equivalent to .NEQV..

Rules for Use

- Parentheses can be used to specify the ordering of an expression evaluation. Within parentheses, and where parentheses do not dictate evaluation order, the order is understood to be as follows:

  Function reference
  Subscript evaluation
  Exponentiation
  Multiplication and division
  Addition and subtraction
  .LT., .LE., .EQ., .NE., .GT., .GE.
  .NOT.
  .AND.
  .OR.
  .EQV., .NEQV., .XOR., .EOR.

- It is invalid to have two contiguous logical operators except when the second operator is .NOT.; however, two consecutive .NOT. operations are not permitted.

  Valid Expressions

  ```
  F .AND. .NOT. D
  M .OR.  .NOT. N
  A .AND. .NOT. (B.OR.C)
  ```

  The latter expression is .TRUE. if and only if A is .TRUE. and both B and C are .FALSE..

Invalid expressions	Error Analysis
A .AND. .OR. B	Two contiguous logical operators
A .NOT. .NOT. B	Two contiguous .NOT. operators

4.5.2 Evaluation of Logical Expressions

All terms of a logical expression are completely evaluated.

4.6 Use of Hollerith Constants in Expressions

A Hollerith constant of one to eight characters can be used as an operand in an assignment statement, relational expression, or arithmetic expression. The value of the Hollerith constant is the ASCII code representation of the character based on the forms:

nHs A left-justified string with trailing blank character padding if necessary.

nRs A right-justified string with leading binary zero padding if necessary.

The data type of the Hollerith constant is determined by its use in the assignment statement, relational expression, or arithmetic expression. The Hollerith constant is treated as the same data type as the other operand involved in the assignment statement or expression. (For double complex, the constant is treated as the real part of a double complex constant; the value of the imaginary part is zero.)

Using Hollerith constants in this manner is allowed primarily to provide compatiblity with earlier uses of FORTRAN. However, it is recommended that the CHARACTER data type be used instead of Hollerith because the CHARACTER data type is more general and tends to be more convenient. In addition, CHARACTER is part of the ANSI-78 standard for FORTRAN; Hollerith is not.

Table 4-2
Hollerith Constants Used with Operators

Type of Other Operand	Number of Leftmost Characters (nHs)	Number of Rightmost Characters (nRs)	Type of Constant
INTEGER*1	1	1	INTEGER*2
INTEGER*2	2	2	INTEGER*2
INTEGER (INTEGER*4)	4	4	INTEGER (INTEGER*4)
INTEGER*8	8	8	INTEGER*8
REAL (REAL*4)	4	4	REAL (REAL*4)
DOUBLE PRECISION (REAL*8)	8	8	DOUBLE PRECISION (REAL*8)
COMPLEX (COMPLEX*8)	8	8	COMPLEX (COMPLEX*8)
COMPLEX*16	8	8	COMPLEX*16 (real part only)

Examples

Variable Type	Statement	Result (hexadecimal)
INT1 is INTEGER*1	INT1 = 1HA	41
IHF is INTEGER*2	IHF = 1HA	4120
IWD is INTEGER (INTEGER*4)	IWD = 2RAB	00004142
INT1 is INTEGER*1	INT1 = 3HABC	41
IHF is INTEGER*2	IF(IHF .EQ. 4HABCD) THEN	Comparison is made on leftmost 16 bits (4142)
REAL is REAL	IF(REAL .GT. 5HABCDE) THEN	Comparison is made on leftmost 32 bits (41424344)
DP is DOUBLE PRECISION	IF(DP .LE. 2RAB) THEN	Comparison is made on the 64 bit constant (0000000000004142)
IHW is INTEGER*2	IHW = 4HABCD+1	4345 (rightmost 16 bits of 41424344 + 1)
INT1 is INTEGER*1	INT1 = 3RAGE-2	43 (rightmost 8 bits of 414745 - 2)
IW is INTEGER (INTEGER*4)	IW = 2HAB+4	41422024 (41422020 + 4)

A Hollerith constant used in conjunction with an arithmetic operator (+,-,*,/, or **) is formatted as follows:

- Form nHs is left justified with trailing blanks and is typed as an integer word constant ($1 \leq n \leq 4$) or as an integer doubleword constant ($5 \leq n \leq 8$).
- Forms 's' * and nRs are right justified with leading zeros and are typed as integer word constants ($1 \leq n \leq 4$) or as integer doubleword constants ($5 \leq n \leq 8$).

* This form of Hollerith constant is valid only if option 10 is set at compile time.

4.7 Use of Hexadecimal, Binary, and Octal Constants in Expressions

Hexadecimal, binary, and octal constants can be used as operands in expressions and argument lists, or they can be used in DATA statement definitions. Such a constant is represented by a string of characters in one of the forms:

Z's' or nZs or X's'	A hexadecimal constant
B's'	A binary constant
O's'	An octal constant

s — A string of hexadecimal characters (0-9 and A-F) for the Z or X forms, a string of binary digits (0-1) for the B form, or a string of octal digits (0-7) for the O form.

n — A positive integer specifying the number of characters in the string.

The constant will be right-justified, zero-filled, and typed according to the context in which it is used.

A hexadecimal, binary, or octal constant used with a replacement operator (e.g., the equal sign =) or any of the relational operators (.LT., .LE., .EQ., .NE., .GE., .GT.) is formatted so that the type of the constant is the same as the type of the other operand. The imaginary part of complex and double complex numbers is assumed to be equal to zero if it is not specified (i.e., without parenthesis).

A hexadecimal, binary, or octal constant used as an initializing value in a DATA statement is formatted into the size appropriate to the item being initialized; however, no type conversion (integer to real or real to integer) is made.

If the hexadecimal, binary, or octal constant is used in conjunction with an arithmetic operator (+, -, *, /, or **), the constant is interpreted as integer. For single and double precision real types, the integer constant is converted to floating point. For very large values, some precision may be lost in the conversion. All floating point types are normalized. Complex constants of the following forms are allowed:

(Z'41100000', Z'BEF00000') is the same as (1.0, - 1.0)

(Z'4110000000000000', Z'BEF0000000000000') is the same (1D0, - 1D0)

4.8 Use of Strings in Argument Lists

The following conventions apply when constructing Hollerith or hexadecimal strings that are used as parameters in subroutine or function calls.

Table 4-3
Hollerith or Hexadecimal String Conventions

String	Designation	Constant Address Type	Constant Form
nHs	s is 1-4 chars	Word	Left-justified/ blank-filled
or 's'*	s is 5-8 chars	Doubleword	Left-justified/ blank-filled
	s is more than 8 chars long	Word	Left-justified/ blank-filled in word array
nRs	s is 1-4 chars	Word	Right-justified/ zero-filled
	s is 5-8 chars	Doubleword	Right-justified/ zero-filled
nZs	s is 1-8 chars	Word	Right-justified/ zero-filled
	s is 9-16 chars	Doubleword	Right-justified/ zero-filled

* This form of Hollerith constant is valid only if option 10 is set at compile time.

CHAPTER 5

ASSIGNMENT STATEMENTS

5.1 General

Assignment statements cause the value of variables, array elements, or character substrings to be set to the results of the evaluation of an expression.

The four types of assignment statements are:

- Arithmetic
- Logical
- Character
- Statement label (ASSIGN)

With the exception of statement label (ASSIGN) statements, assignment statements have the form:

```
v=e
```

v	Any variable, array element, or character substring.
e	An arithmetic, logical, or character expression.

The equal sign denotes replacement rather than equality. Thus, an assignment statement causes evaluation of the expression on the right of the equal sign and the placement of the result in the storage space allocated to the variable or array element on the left of the equal sign.

Rules for Use

- If e is an arithmetic expression, v must be a variable or array element of type integer, real, double precision, or complex.
- If e is a logical expression, v must be a logical variable or array element.
- If e is a character expression, v must be a character variable, character array element, or character substring.
- Both v and the equal sign must appear on the same line even when the statement is part of a logical IF statement. Moreover, the line containing v= must be the initial line of the statement unless the statement is part of a logical IF statement; in such a case the v= must occur on the same line as the logical IF or on the line immediately following it.

5.2 Arithmetic Assignment Statements

An arithmetic assignment statement assigns the value of the expression on the right of the equal sign to the numeric variable or array element on the left of the equal sign.

The arithmetic assignment statement has the form:

```
v = e
```

v An arithmetic variable or array element.

e An arithmetic expression.

Rules for Use

- The data types of v and e can be integer, real, double precision, or complex.
- Values must have been previously assigned to all symbolic references in the expression.
- The expression must yield an arithmetic value.

Conversion Rules

- If the data type of v and e differ, the expression is evaluated and the result is converted to the type of v. The following chart details the required conversions.

Variable (v) Type	Expression (e) Type	Description of Conversions
INTEGER*1	INTEGER*1 (IB)	No conversion required
	INTEGER*2 (IH)	Bits 8-15 of IH are stored into v
	INTEGER or INTEGER*4 (IW)	Bits 24-31 of IW are stored into v
	INTEGER*8 (ID)	Bits 56-63 of ID are stored into v
	REAL or REAL*4 (R)	R is converted to an IW; bits 24-31 of IW are stored into v
	DOUBLE PRECISION or REAL*8 (DP)	DP is converted to an ID; bits 56-63 of ID are stored into v
	COMPLEX or COMPLEX*8 (CW)	The real part of CW is converted to an IW; bits 24-31 of IW are stored into v
	COMPLEX*16 (CD)	The real part of CD is converted to an ID; bits 56-63 of ID are stored into v

Variable (v) Type	Expression (e) Type	Description of Conversions
INTEGER*2	INTEGER*1 (IB)	IB is stored into bits 8-15 of v; zeros are stored into bits 0-7
	INTEGER*2 (IH)	No conversion required
	INTEGER or INTEGER*4 (IW)	Bits 16-31 of IW are stored into v
	INTEGER*8 (ID)	Bits 48-63 of ID are stored into v
	REAL or REAL*4 (R)	R is converted to an IW; bits 16-31 of IW are stored into v
	DOUBLE PRECISION or REAL*8 (DP)	DP is converted to an ID; bits 48-63 of ID are stored into v.
	COMPLEX or COMPLEX*8 (CW)	The real part of CW is converted to an IW; bits 16-31 of IW are stored into v
	COMPLEX*16 (CD)	The real part of CD is converted to an ID; bits 48-63 of ID are stored into v
INTEGER or INTEGER*4	INTEGER*1 (IB)	IB is stored into bits 24-31 of v; zeros are stored into bits 0-23
	INTEGER*2 (IH)	IH is stored into bits 16-31 of v; zeros are stored into bits 0-15
	INTEGER or INTEGER*4 (IW)	No conversion required
	INTEGER*8 (ID)	Bits 32-63 of ID are stored into v
	REAL or REAL*4 (R)	R is converted to an IW and then stored into v
	DOUBLE PRECISION or REAL*8 (DP)	DP is converted to an ID; bits 32-63 of ID are stored into v
	COMPLEX or COMPLEX*8 (CW)	The real part of CW is converted to an IW and then stored into v
	COMPLEX*16 (CD)	The real part of CD is converted to to an ID; bits 32-63 of ID are stored into v

Variable (v) Type	Expression (e) Type	Description of Conversions
INTEGER*8	INTEGER*1 (IB)	IB is stored into bits 56-63 of v; zeros are stored into bits 0-55
	INTEGER*2 (IH)	IH is stored into bits 48-63 of v; zeros are stored into bits 0-47
	INTEGER or INTEGER*4 (IW)	IW is stored into bits 32-63 of v; zeros are stored into bits 0-31
	INTEGER*8 (ID)	No conversion required
	REAL or REAL*4 (R)	R is converted to a DP; DP is converted to an ID and stored into v
	DOUBLE PRECISION or REAL*8 (DP)	DP is converted to an ID and stored into v
	COMPLEX or COMPLEX*8 (CW)	The real part of CW is converted to a DP; DP is converted to an ID and stored into v
	COMPLEX*16 (CD)	The real part of CD is converted to an ID and stored into v
REAL or REAL*4	INTEGER*1 (IB)	IB is converted to an IW; IW is converted to an R and stored into v
	INTEGER*2 (IH)	IH is converted to an IW; IW is converted to an R and stored into v
	INTEGER or INTEGER*4 (IW)	IW is converted to an R and stored into v
	INTEGER*8 (ID)	ID is converted to an IW; IW is converted to an R and stored into v
	REAL or REAL*4 (R)	No conversion required
	DOUBLE PRECISION or REAL*8 (DP)	Bits 0-31 of DP are stored into v
	COMPLEX or COMPLEX*8 (CW)	The real part of CW is stored into v

Variable (v) Type	Expression (e) Type	Description of Conversions
	COMPLEX*16 (CD)	Bits 0-31 of the real part of CD are stored into v
DOUBLE PRECISION or REAL*8	INTEGER*1 (IB)	IB is converted to an ID; ID is converted to a DP and stored into v
	INTEGER*2 (IH)	IH is converted to an ID; ID is converted to a DP and stored into v
	INTEGER or INTEGER*4 (IW)	IW is converted to an ID; ID is converted to a DP and stored into v
	INTEGER*8 (ID)	ID is converted to a DP and stored into v
	REAL or REAL*4 (R)	Bits 0-31 of R are stored into bits 0-31 of DP; zeros are stored into bits 32-63
	DOUBLE PRECISION or REAL*8 (DP)	No conversion required
	COMPLEX or COMPLEX*8 (CW)	The real part of CW is converted to a DP and stored into v
	COMPLEX*16 (CD)	The real part of CD is stored into v
COMPLEX or COMPLEX*8	INTEGER*1 (IB)	IB is converted to an IW; IW is converted to an R and stored into the real part of CW; zeros are stored into the imaginary part of CW
	INTEGER*2 (IH)	IH is converted to an IW; IW is converted to an R and stored into the real part of CW; zeros are stored into the imaginary part of CW
	INTEGER or INTEGER*4 (IW)	IW is converted to an R and stored into the real part of CW; zeros are stored into the imaginary part of CW

Variable (v) Type	Expression (e) Type	Description of Conversions
	INTEGER*8 (ID)	ID is converted to an IW; IW is converted to an R and stored into the real part of CW; zeros are stored into the imaginary part of CW
	REAL or REAL*4 (R)	R is stored into the real part of CW; zeros are stored into the imaginary part of CW
	DOUBLE PRECISION or REAL*8 (DP)	DP is converted to an R and stored into the real part of CW; zeros are stored into the imaginary part of CW
	COMPLEX or COMPLEX*8 (CW)	No conversion required
	COMPLEX*16 (CD)	Bits 0-31 of the real part of CD are stored into the real part of CW; bits 0-31 of the imaginary part of CD are stored into the imaginary part of CW
COMPLEX*16	INTEGER*1 (IB)	IB is converted to an ID; ID is converted to a DP and stored into the real part of CD; zeros are stored into the imaginary part of CD
	INTEGER*2 (IH)	IH is converted to an ID; ID is converted to a DP and stored into the real part of CD; zeros are stored into the imaginary part of CD
	INTEGER or INTEGER*4 (IW)	IW is converted to an ID; ID is converted to a DP and stored into the real part of CD; zeros are stored into the imaginary part of CD
	INTEGER*8 (ID)	ID is converted to a DP and stored into the real part of CD; zeros are stored into the imaginary part of CD

Variable (v) Type	Expression (e) Type	Description of Conversions
	REAL or REAL*4 (R)	R is converted to a DP and stored into the real part of CD; zeros are stored into the imaginary part of CD
	DOUBLE PRECISION or REAL*8 (DP)	DP is stored into the real part of CD; zeros are stored into the imaginary part of CW
	COMPLEX or COMPLEX*8 (CW)	Bits 0-31 of the real part of CW are stored into bits 0-31 of the real part of CD and zeros are stored into bits 32-63. Bits 0-31 of the imaginary part of CW are stored into bits 0-31 of the imaginary part of CD and zeros are stored into bits 32-63
	COMPLEX*16 (CD)	No conversion required

Examples of valid arithmetic assignment statements and their descriptions follow. I and J are INTEGER variables; A, B, C, and D are REAL variables; E is a COMPLEX variable.

Statement	Description
A = B	The value of A is replaced by the current value of B.
I = B	The value of B is truncated to an integer value which replaces the value of I.
B = I	The value of I is converted to a real value that replaces the value of B.
E = I**J+D	I is raised to the power J and the result is converted to a real value, to which the value of D is added. This result replaces the real part of the complex variable E. The imaginary part of the complex variable is set to zero.
D = E	The real part of the complex variable E replaces the value of D.
E = D	The value of D replaces the value of the real part of the complex variable E; the imaginary part is set to zero.

5.3 Logical Assignment Statements

A logical assignment statement assigns the value of the logical expression on the right of the equal sign to the variable or array element on the left of the equal sign.

The logical assignment statement has the form

```
v=e
```

v A logical variable or array element.

e A logical expression.

Rules for Use

- Values, either numeric or logical, must have been previously assigned to all symbolic references in the expression.
- The expression must yield a logical value.

Conversion Rules

- If the data type of v and e differ, the expression is evaluated and the result is converted to the type of v. The following chart details the required conversions.

Variable (v) Type	Expression (e) Type	Description of Conversions
(Logical) BIT	BIT (LBIT)	No conversion required
	LOGICAL*1 (LB)	Bit 7 of LB is stored into LBIT
	LOGICAL or LOGICAL *4 (LW)	Bit 31 of LW is stored into LBIT
LOGICAL*1	BIT (LBIT)	LBIT is stored into bits 0-7 of v
	LOGICAL*1 (LB)	No conversion required
	LOGICAL or LOGICAL*4 (LW)	Bits 24-31 of LW are stored into v
LOGICAL or LOGICAL*4	BIT (LBIT)	LBIT is stored into bits 24-31 of v; bits 0-23 are unchanged
	LOGICAL*1 (LB)	LB is stored into bits 24-31 of v; bits 0-23 are unchanged
	LOGICAL or LOGICAL*4 (LW)	No conversion required

Examples of valid logical assignment statements and their descriptions follow; G and L are logical variables; I is an integer variable.

Statement	Description
G = .TRUE.	The value of G is replaced by the logical value .TRUE..
L = .NOT. G	If G is .TRUE., the value of L is replaced by the logical value .FALSE.. If G is .FALSE., the value of L is replaced by the logical value .TRUE..
G = 4. .LT. I	The value of I is converted to a real value; if the real constant 4. is less than this result, the logical value .TRUE. replaces the value of G. If 4. is not less than I, the logical value .FALSE. replaces the value of G.

5.4 Character Assignment Statement

The character assignment statement assigns the value of the character expression on the right of the equal sign to the character variable, array element, or substring on the left of the equal sign.

The character assignment statement has the form

v=e

v A character variable, array element, or substring.

e A character expression.

Character expressions are interpreted as strings of characters. The value of v is determined by evaluating the expression and replacing v with the value of the expression. This value is affected only by the possible extension to the right by blanks or truncation.

Rules for Use

- None of the character positions being replaced in v can be referenced in e; thus, the following assignment is incorrect.

```
STRING (1:10) = STRING (5:14)
```

- Values must have been previously assigned to all symbolic references in the expression.
- Assigning a value to a character substring does not affect character positions in the character variable or array element that are not included in the substring. A character position that is undefined or has a previously assigned value remains unchanged if it is outside the substring.

Conversion Rules

- If the length of the character expression value is greater than the length of v, the character expression value is truncated on the right.
- If the length of the character expression value is less than the length of v, the character expression value is filled on the right with blanks.

Examples of valid character assignment statements follow. Assume that all variables and arrays in the examples are of CHARACTER data type.

Statement	Description
FILE = 'PROG2'	The value of FILE is replaced by 'PROG2'.
CODE (1) = 'MAR'//'COM'	The value of array CODE element 1 is replaced by 'MARCOM'.
TERM(3:10) = 'POSTDATE'	The value of TERM character positions 3 through 10 are replaced by 'POSTDATE'.

5.5 ASSIGN Statement

The ASSIGN statement assigns a statement label value to an integer variable. The variable can then be used to specify a transfer destination in an assigned GO TO statement or to specify a FORMAT statement that is to be used in an input/output statement.

Syntax

```
ASSIGN i TO m
```

i — A statement label of an executable statement or a FORMAT statement in the same program unit.

m — Either an INTEGER (INTEGER*4) or INTEGER*8 variable.

The ASSIGN statement is similar to an arithmetic assignment statement with one exception: the variable becomes defined for use as a statement label reference and becomes undefined as an integer variable.

The ASSIGN statement must be executed before any assigned GO TO or input/output statement(s) in which the assigned variable is to be used. The ASSIGN statement and any assigned GO TO or input/output statement(s) that reference the ASSIGN variable must occur in the same program unit.

For example,

```
ASSIGN 100 TO NUMBER
```

The preceding statement associates the variable NUMBER with statement label 100. As in the following statement, arithmetic operations on the variable become invalid, since arithmetic operations on a statement label are undefined.

```
NUMBER = NUMBER+1
```

The following statement disassociates NUMBER from statement 100, assigns it an integer value 10, and returns its status to an integer variable.

```
NUMBER = 10
```

An assigned GO TO can no longer use the variable NUMBER.

Examples

```
ASSIGN 10 TO NSTART
```

KSTOP, ERROR, and NSTART must be INTEGER (INTEGER*4) or INTEGER*8 variables

```
ASSIGN 99999 TO KSTOP

ASSIGN 250 TO ERROR
```

5.6 Multiple Assignment Statements

A multiple assignment statement assigns a value to more than one variable. The form of the statement is:

$v_1 = v_i \ldots = v_i = e$

v_i Variable or array elements, including character variables, character array elements, and character substrings. Character data cannot be mixed with other types of data.

e An expression.

Replacement is performed from right to left. Type conversion is performed, if necessary, across each equal sign. Therefore, you can control the number and order of conversions.

```
RR = I = R = D = 1.4576342198762D0
```

is equivalent to

```
D = 1.4576342198762D0
R = D
I = R
RR = I
```

performed in that order.

Therefore, if RR and R are REAL, I is INTEGER, and D is DOUBLE PRECISION, then the assigned values would be

```
D = 1.4576342198762D0
R = 1.4576342
I = 1
RR = 1.0
```

Note that even though RR and R are both real, they are not assigned the same value, since replacement proceeds from right to left.

Execution of a multiple character assignment statement causes the character expression e to be evaluated. The character entities $v_1, \ldots, v_i$ in turn, taken from right to left, are defined and assigned the value of e. This value is affected only by possible truncations and extensions to the right by blanks, depending on the sizes of the intermediate entities v_i. For example, if A, B, and C are CHARACTER *6, *3, and *5, respectively, then:

```
A = B = C = 'QUOTE'
```

will result in

```
C = 'QUOTE'
B = 'QUO'
A = 'QUO␣␣␣'
```

5.7 Full Array Assignments

A full array assignment statement may be used to assign a value to every element of an array. The statement is of the form:

```
a = e
```

a — An array name.

e — An expression.

For example,

```
TABLE = 1.0
```

would initialize each element in the entire array TABLE to the value 1.0. Notice that if the statement is altered to an array element assignment, such as:

```
TABLE (3) = 1.0
```

only element three (3) of array TABLE receives the value 1.0. All other elements of the array are unaffected.

CHAPTER 6

CONTROL STATEMENTS

6.1 General

Control statements are executable statements that determine the order in which other statements are executed. Execution normally begins with the first executable statement and proceeds sequentially through successive statements. Control statements permit alteration of this normal process and allow the user to transfer control to a point within the same program unit or to another program unit. These statements also govern iterative processing, suspension of program execution, and program termination.

The control statements described in this chapter are:

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO
- Arithmetic IF
- Logical IF
- IF THEN
- ELSE IF THEN
- ELSE
- END IF
- DO
- DO forever
- DO UNTIL
- DO WHILE
- LEAVE
- CONTINUE
- END DO
- SELECT CASE
- CASE
- END SELECT
- STOP
- END
- PAUSE

The CALL and RETURN control statements are discussed in Chapter 9.

6.2 GO TO Statements

GO TO statements transfer control to an executable statement specified by a statement label in the GO TO statement. Control is transferred either to the same statement each time the GO TO is executed or to one of a set of statements based on the value of an expression.

The three types of GO TO statements are:

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

6.2.1 Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the same statement each time it is executed.

Syntax

```
GO TO x
```

x — The statement label of an executable statement within the same program unit.

Example

```
      GO TO 376
310   V(7)= ABAR-V(10)
            .
            .
            .
376   V(5)=ZOT
      GO TO 310
```

In the preceding example, statement 376 may be executed before statement 310 even though it appears after statement 310 in the text.

6.2.2 Computed GO TO Statement

The computed GO TO statement transfers control to a statement based on the value of an integer expression within the statement.

Syntax

GO TO (x_1 [,x_n]. . .) [,] i

x_n Labels of executable statements within the same program unit.

i An INTEGER*1, INTEGER*2, or INTEGER (INTEGER*4) expression.

Control is transferred to the ith statement label depending on the current value of i.

Rule for Use

If the value of i is less than one or greater than n, control proceeds to the first executable statement after the computed GO TO.

Example

```
        J=3
        .
        .     .
        .
        GO TO (115, 306, 700, 8019, 73),J
99      CONTINUE
        .
        .
        .
```

In the preceding example, when J is equal to 1, the computed GO TO transfers control to statement 115; when J is equal to 2, control is transferred to statement 306, and so on. If J were less than 1 or greater than 5, control would fall through to the statement following the computed GO TO (statement 99).

6.2.3 Assigned GO TO Statement

The assigned GO TO statement transfers control to a statement label that is represented by a variable. The relationship between the variable and the specific statement label must be established by an ASSIGN statement. Thus, the transfer destination can be changed depending on the most recently executed ASSIGN statement.

Syntax

GO TO i [,] [(x [,x_n] . . .)]

i — An INTEGER (INTEGER*4) or INTEGER*8 variable name.

x_n — Labels of executable statements within the same program unit.

Rules for Use

- When a list of statement labels is specified, the value of i should have been assigned to one of the labels in the list. However, no check is made to verify this assignment.
- The execution of an ASSIGN statement specifying a variable name must logically precede the execution of an assigned GO TO that references that name.

Example

```
ASSIGN 50 TO IERROR
    .
    .
    .
GO TO IERROR
    .
    .
    .
ASSIGN 90 TO LOCAL
    .
    .
    .
GO TO LOCAL, (70, 90, 110)
```

6.3 IF Statements

IF statements transfer control to one of a series of statements depending upon a specified condition. The three types of IF statements are:

- Arithmetic
- Logical
- Block

6.3.1 Arithmetic IF Statement

The arithmetic IF statement transfers control to one of three statements based upon the value of an arithmetic expression.

Syntax

IF (e) x_1,x_2,x_3

e — An arithmetic expression.

x_1,x_2,x_3 — Labels of executable statements within the same program unit.

Control is transferred to statement label x_1, x_2, or x_3 depending on whether the value of the arithmetic expression e is less than, equal to, or greater than zero, respectively.

Rules for Use

- The data type of e can be integer, real, or double precision; it must not be complex.
- All three statement labels are required; however, they need not refer to different statements.

Examples

```
IF (ZAP) 2,3,4
```

In the preceding example, when ZAP < 0, control is transferred to statement 2; when ZAP=0, control is transferred to statement 3; when ZAP > 0, control is transferred to statement 4.

```
IF (NUMBER) 2,2,4
```

In the preceding example, when NUMBER $\leq$ 0, control is transferred to statement 2; when NUMBER > 0, control is transferred to statement 4.

```
IF (AMTX (3,1,2))7,2,1
```

In the preceding example, when the value of the element (3, 1, 2) of array AMTX is less than 0, control is transferred to statement 7; when the value is equal to 0, control is transferred to statement 2; when the value is greater than 0, control is transferred to statement 1.

6.3.2 Logical IF Statement

The logical IF statement conditionally executes a single FORTRAN statement. Execution of the statement is based on the value of a logical expression within the logical IF statement.

Syntax

IF (e) s

e A logical expression.

s An executable statement.

The logical expression e in the IF statement is evaluated first. If the value of the expression is .TRUE., the statement s is executed. If the value of the expression is .FALSE., control is transferred to the next executable statement after the logical IF; the statement s is not executed.

Rules for Use

- The executable statement s must not be a DO, END, block IF, ELSE IF, ELSE, END IF, END DO, SELECT CASE, CASE, END SELECT, or another logical IF statement.
- If s is an assignment statement, the left side of the assignment and the equal sign (=) must be on the same line, either immediately following the IF (e) or on a continuation line with all blanks following the IF (e) (refer to statements 4 and 5 in the following example).

Example

```
      LOGICAL Q,R,Z
1     IF (I.LT.20) GO TO 115
2     IF (Q.AND.R) ASSIGN 10 TO J
3     IF (Z) CALL DECL (A,B,C)
4     IF (A .GT. 10 .OR. B .LE. P1/2)I=J
5     IF (R .GT. 12 .OR. A .LE. 15)
      X K=M
```

6.3.3 Block IF Construct

Block IF constructs conditionally execute blocks (or groups) of statements. The statements associated with the block IF are:

```
IF (e) THEN
ELSE IF (e) THEN
ELSE
END IF
```

The block IF construct has the form

```
IF (e1) THEN
  block

[ELSE IF (e2) THEN
  block]
   .
   .
   .
[ELSE
 block]

END IF
```

e_i A logical expression.

block A sequence of zero or more complete FORTRAN 77+ statements. (This sequence is called a statement block.)

Each statement in a block IF construct, except the END IF statement, has an associated statement block. The statement block consists of all the statements following the IF statement up to (but not including) the next IF statement in this block IF construct. The statement block is conditionally executed based on the values of logical expressions in the preceding IF statements.

6.3.3.1 IF THEN Statement

The IF THEN statement begins a block IF construct. The block following is executed if the value of the logical expression in the IF THEN statement is true, as indicated in Figure 6-1.

Syntax

```
IF (e) THEN
```

e A logical expression.

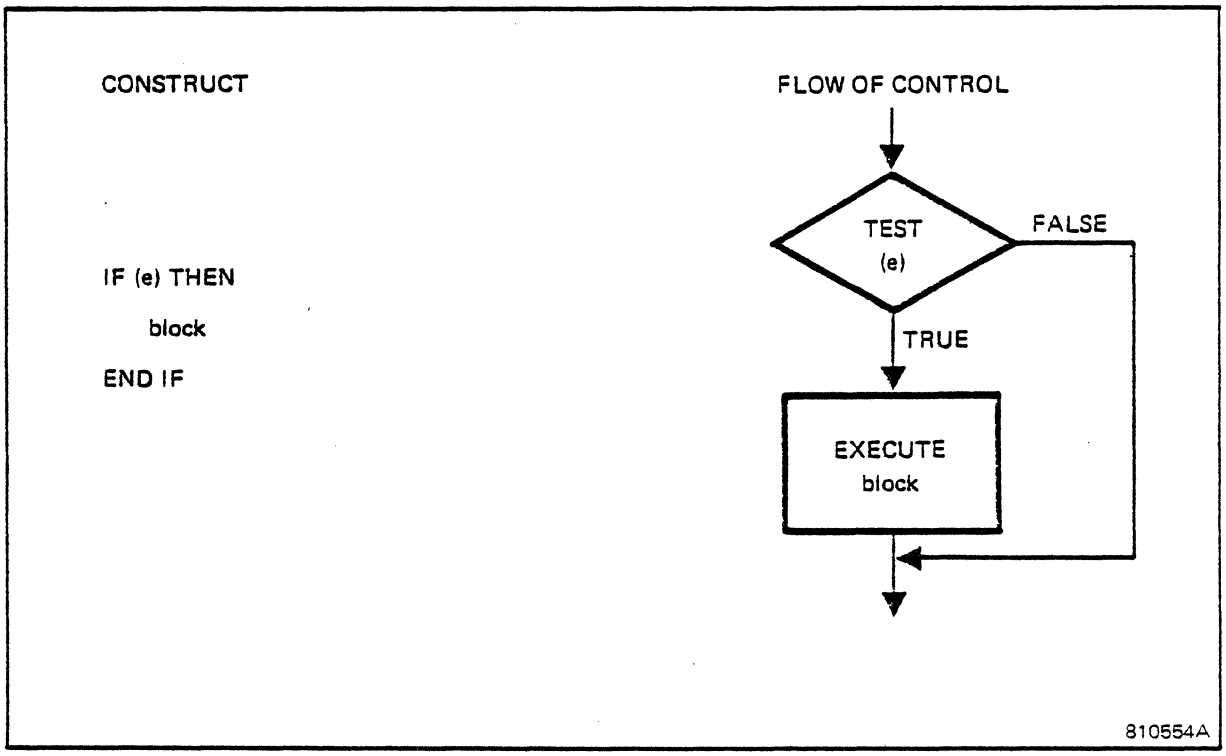


Figure 6-1. Simple Block IF Construct

Rule for Use

Transfer of control into an IF block from outside the IF block is prohibited.

Example

```
100   IF (A.LT.B) THEN
            DIFF=B-A
            PRINT 300, A,B,DIFF
300   FORMAT ('A= ', F6.2, 'B= ', F6.2, 'DIFFERENCE= ', F6.2)
      END IF
```

In the preceding example, the statement block consists of all the statements between the IF THEN and the END IF statements.

The logical expression A.LT.B is evaluated first. If the value of the expression is true, the statement block is executed. If the value of the expression is false, control passes to the END IF; the block is not executed.

6.3.3.2 ELSE IF THEN Statement

The ELSE IF THEN statement is optional. It specifies a statement block to be executed if the value of the logical expression in the statement is true and no preceding statement block in the block IF construct has been executed as indicated in Figure 6-2.

Syntax

ELSE IF (e) THEN

e A logical expression.

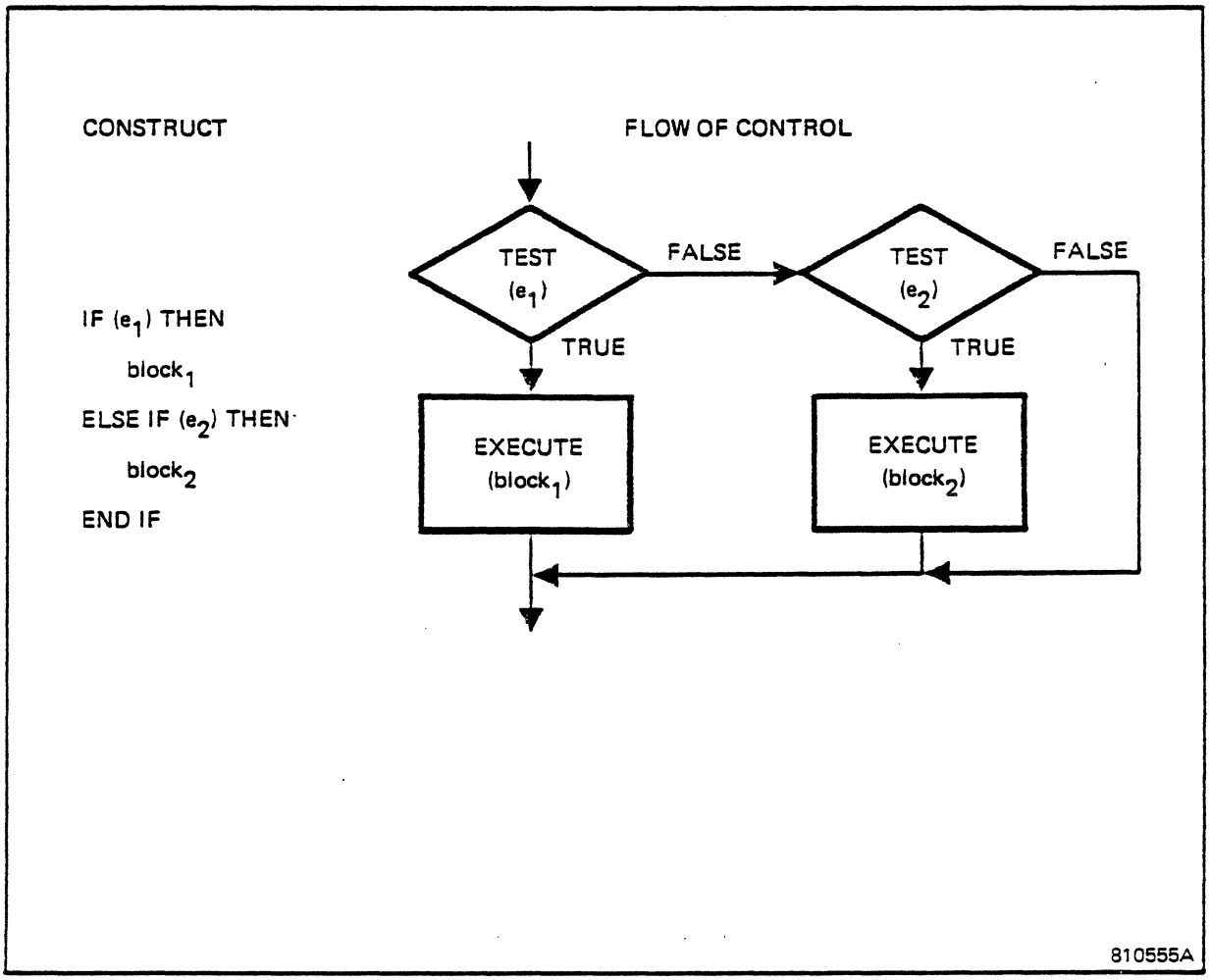


Figure 6-2. Block IF Construct with ELSE IF THEN Statement Block

A block IF construct can contain any number of ELSE IF THEN statements, as indicated in Figure 6-3.

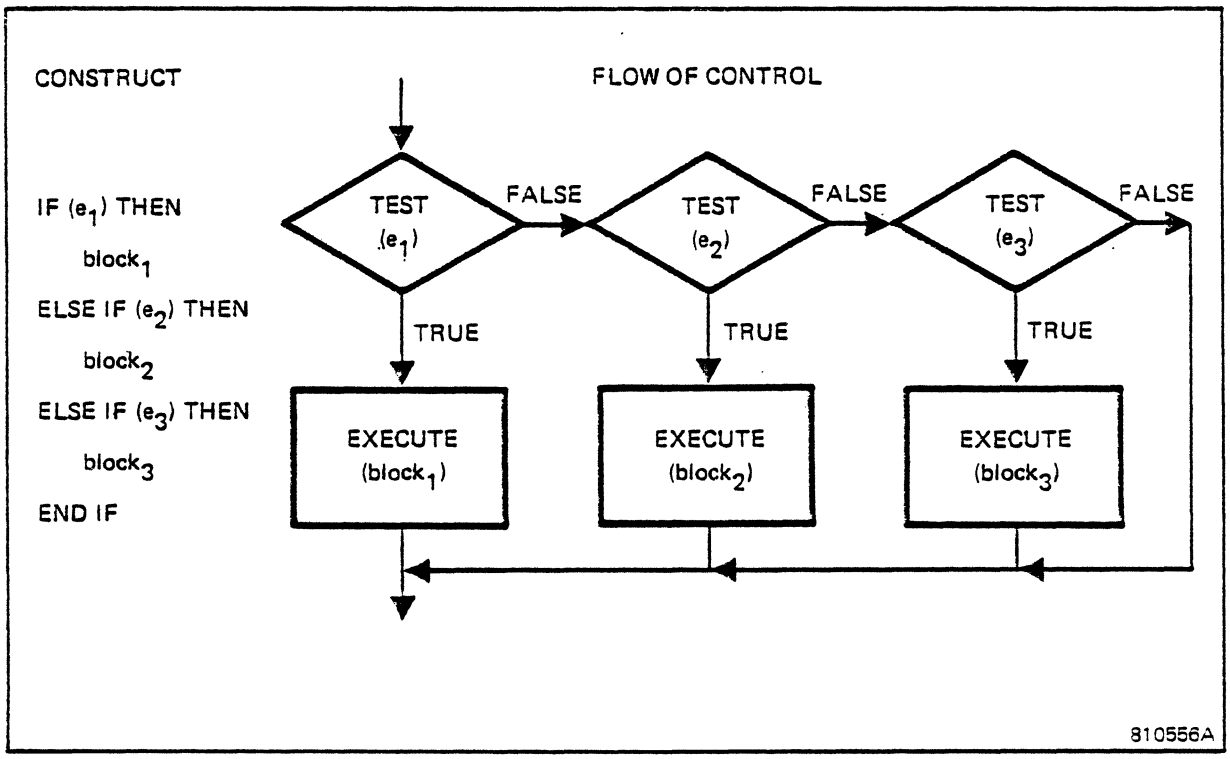


Figure 6-3. Block IF Construct With Multiple ELSE IF THEN Statement Blocks

Rules for Use

- Transfer of control into an ELSE IF THEN block from outside the block is prohibited.
- The statement label, if any, of an ELSE IF THEN statement must not be referenced by any statement.
- An ELSE IF THEN block can be empty.

Example

```
100     IF (A.EQ.B) THEN
            PRINT 200,A
200         FORMAT ('A AND B=', F6.2)
        ELSE IF (A.LT.B) THEN
            DIFF=B-A
            PRINT 300, A,B,DIFF
300         FORMAT ('A=',F6.2,'B=',F6.2,'DIFFERENCE=',F6.2)
        END IF
```

In the preceding example, block 1 consists of all the statements between the IF THEN and the ELSE IF THEN statements; block 2 consists of all the statements between the ELSE IF THEN and the END IF statements. Execution proceeds as follows:

1. If A is equal to B, block 1 is executed.
2. If A is not equal to B but A is less than B, block 2 is executed.
3. If A is not equal to B and A is not less than B, neither block 1 nor block 2 is executed; control transfers to the END IF statement.

6.3.3.3 ELSE Statement

The ELSE statement specifies a statement block to be executed if no preceding statement block in the block IF construct has been executed (see Figure 6-4).

Syntax

```
ELSE
```

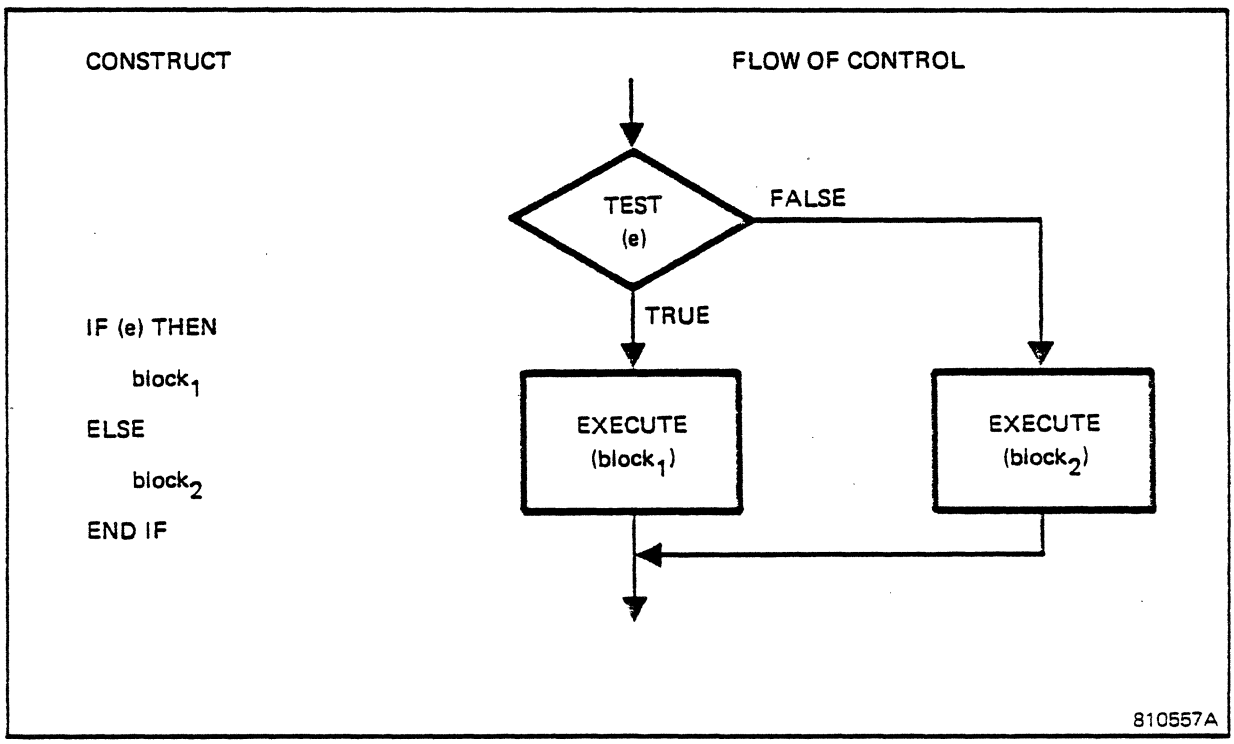


Figure 6-4. Block IF Construct with ELSE Statement Block

Rules for Use

- An ELSE block must end with an END IF statement.
- Transfer of control into an ELSE block from outside the ELSE block is prohibited.
- The statement label, if any, of an ELSE statement must not be referenced by any statement.

Example

```
100    IF(AD.LT.TR) THEN
          COUNT=COUNT + 1
          TERM(COUNT) = AD
       ELSE
          NEW = NEW + 1
       END IF
```

In this example, block 1 consists of all the statements between the IF THEN and ELSE statements, and block 2 consists of all the statements between the ELSE and the END IF statements. Execution proceeds as follows:

1. If the value of AD is less than TR, block 1 is executed.
2. If the value of AD is equal to or greater than TR, block 2 is executed.

6.3.3.4 END IF Statement

The END IF statement is used to end a block IF construct.

Syntax

```
END IF
```

Rule for Use

For each block IF construct, there must be a corresponding END IF statement in the same program unit.

Example

```
IF (B.LT.SO) THEN
  .
  .
  .
ELSE IF (VAL .LE.20) THEN
  .
  .
  .
ELSE
  .
  .
  .
END IF
```

6.3.3.5 Nested Block IF Construct

A block IF construct can be included in a statement block of another block IF construct. However, the nested block IF construct must be completely contained within a statement block; it must not overlap statement blocks. Similarly, a block IF construct can be included in the range of a DO statement if the complete construct is written within the DO range.

Form

```
IF (e) THEN
          ┌──────────────
          │ IF (e) THEN
          │    blocka
blockl    │ ELSE
          │    blockb
          │
          │ END IF
          └──────────────
ELSE
          block2
END IF
```

Example

```
      IF (B.LT.50) THEN
50          GO = GO + 1
100         IF (VAL .LE. 10) THEN
               TOT = TOT +1
            ELSE
200            UNTOT = UNTOT + 1

            END IF
300   ELSE
            OUTGO = OUTGO + 1
400   END IF
      .     .
      .
      .
500   END
```

In the preceding example, blockl contains a nested block IF construct. Execution proceeds as follows:

1. If B is less than 50, blockl is executed. If the value of VAL is less than or equal to 10, blocka is executed. If the value of VAL is greater than 10, blockb is executed.

2. If B is greater than or equal to 50, block2 is executed; the nested IF construct is not executed, because it is not in block2.

6.3.3.6 IF Level

Awareness of the IF level of a statement helps insure that block IF constructs are completely contained within a statement block.

The IF level of a statement s is defined as:

n1-n2

n1 The number of block IF statements from the beginning of the program unit, up to and including s.

n2 The number of END IF statements in the program unit, up to but not including s.

Rules for Use

- The IF level of every statement must be zero or positive, i.e., there must be at least as many block IF statements to and including a given statement as there are END IF statements that precede it. In the previous example (section 6.3.3.5), the following IF levels apply:

Statement Number	IF Level
50	1
100	2
200	2

- The IF level of each block IF, ELSE IF, ELSE, and END IF statement must be positive (refer to the example in section 6.3.3.5).

Statement Number	IF Level
100	2
300	1
400	1

- The IF level of the END statement of each program unit must be zero, i.e., there must be one block IF statement for every END IF statement (refer to the example in section 6.3.3.5).

Statement Number	IF Level
500	0

6.4 DO Statement

The DO statement specifies iterative processing; statements in its range are executed repeatedly a specified number of times.

Syntax

DO [x[,]] i= m_1, m_2 [, m_3]

x — The statement label of an executable statement. x is optional when END DO is used as the terminal statement.

i — A variable of any integer, real or double precision data type.

m_1, m_2, m_3 — The initial, terminal, and incremental parameters, respectively. Each is an arithmetic expression with a data type of integer, real, or double precision. The default value of m_3 is one.

The expressions m_1, m_2, and m_3 are evaluated first to determine values for the initial, terminal, and incremental parameters. The value of the initial parameter m_1 is assigned to the control variable i, and the executable statements in the range of the DO loop are executed. During the second execution, i = m_1 + m_3; during the third, i = m_1 + 2m_3, etc., until i is out of the range of m_1 to m_2.

Rules for Use

- The terminal statement of a DO loop is identified either by the label x that appears in the DO statement, or by an END DO if x is not specified.
- The DO, a comma, and the equal sign must appear on the initial line of a DO statement. Thus, either of the following is correct:

```
      DO 33 I = 40,                    DO 33, I =
     * 60,2              or           * 40,60,2
```

- It is invalid to transfer control into the range of a DO loop from outside the loop.
- Within the range of a DO statement, there may be other DO statements, in which case the DO statements are said to be nested.

6.4.1 Range of the DO Statement

The range of the DO statement includes all executable statements following the DO statement that specifies the DO loop, up to and including the terminal statement.

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

```
        IF      (A.GT.B) THEN
                DO   50 L=10,250,2
                  .
                  .
                  .
50              E(T)= D(L)+F
        END IF
```

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

```
        DO 50 L=10,250,2
        IF (A.GT.B) THEN
                D=B
                F=A-B
        ELSE
                D=0.0
                F=0
        END IF
50      E(T) = D(L)+F
```

6.4.2 Active and Inactive DO Loops

A DO loop is initially inactive. It becomes active only when its DO statement is executed.

Once active, the DO loop becomes inactive only when:

- Its iteration count is tested 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.
- A STOP statement in the executable program is executed, or execution is terminated for any other reason (error conditions).

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 specified in a CALL statement to a statement that is not in the range of the DO loop.

6.4.3 Terminal Statement of the DO

The terminal statement of a DO loop is identified by the label that appears in the DO statement, or it may be an END DO statement. The following rules govern the use of the terminal statement:

- The terminal statement must be an executable statement.
- The terminal statement must physically follow its associated DO.
- The terminal statement must not be an arithmetic IF, unconditional or assigned GO TO, RETURN, STOP, PAUSE, END, or another DO; also, it may not be a block IF, ELSE IF, ELSE, END IF, SELECT CASE, CASE, END SELECT, or LEAVE statement.
- If the terminal statement is a logical IF statement, it can contain any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, SELECT CASE, CASE, END CASE, or another logical IF.
- If the terminal statement is a logical IF and its expression value is .FALSE., the statement in the logical IF is not executed before the statements in the DO range are repeated.
- If the terminal statement is a logical IF and its expression is .TRUE., the statement of the logical IF is executed, after which the statements in the DO range are reiterated unless the statement contained in the logical IF statement alters the flow of control.
- The terminal statement of the inner DO of a nested DO loop may also be the terminal statement of the outer DO. Note however, that an END DO used as a terminal statement terminates only the preceding DO loop.
- If the terminal statement is shared, it is "owned" by the innermost active DO loop.

6.4.4 Index of the DO

The controlling variable i is called the index of the DO range. The index is negative, zero, or positive, depending upon the evaluation of the initial and terminal parameters m_1 and m_2.

The following rules govern the use of the index and associated parameters:

- The terminal parameter need not be greater than or equal to the initial parameter; m_3 can be negative.
- Evaluation of m_1, m_2, and m_3 (when represented by arithmetic expressions) occurs only once at the beginning of execution of the DO statement.
- m_3 is optional; the default is 1.
- The DO index may not be redefined during execution of the range of the DO loop.

6.4.5 DO Iteration Control

The number of times a DO loop is executed is called the trip count, which is determined as

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

The value of the count establishes the number of times the loop range is executed. If the count is zero, the loop immediately becomes inactive and its range is not executed.

Users may set compiler option 11, which causes all DO loops to execute at least once.

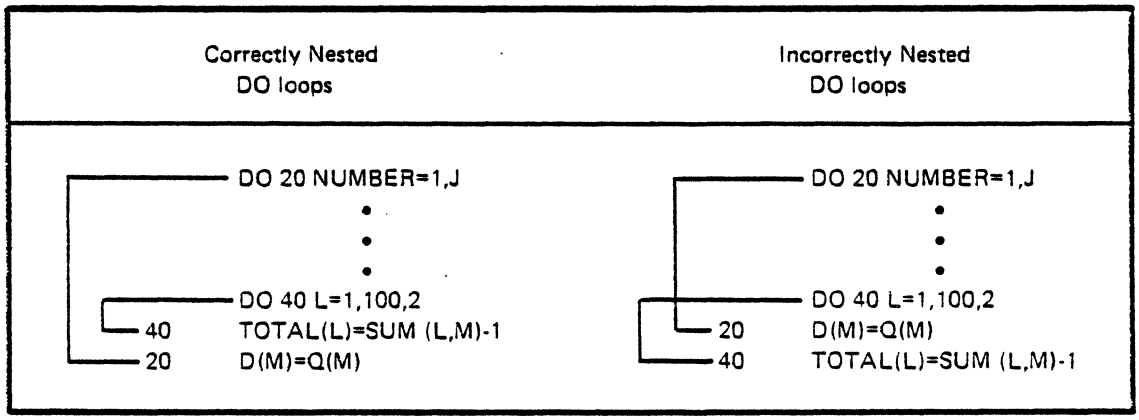
The trip count is determined at the time the DO statement is executed. Entities that appear in the expression used to calculate this value may be changed during execution of a loop without affecting the number of times a loop is executed.

The user should note that the trip count calculation itself must not cause arithmetic overflow, or unpredictable results will occur. The maximum allowed trip count value is $(2^{31}-1)$.

When a DO loop has been executed, the DO index variable retains a value that is the value it would have had in the next execution of the loop. However, premature exiting of a DO loop by a LEAVE, GOTO, or call to a procedure that has an alternate return causes the index variable to retain the value that was current when the loop was exited.

6.4.6 Nested DO Loops

A DO loop can contain one or more complete DO loops. The range of an inner nested DO loop must be completely contained within the range of the next outer loop. Figure 6-5 illustrates nested loops.



810558A

Figure 6-5. Nested DO Loops

Nested DO loops can share terminal statements. For example,

```
      DIMENSION A (15,15) , B(15), C(15)
        .
        .
        .
      DO 73  K=1,15
      C(K) = 0.0
      DO 73  J=1,15
73    C(K) = C(K) + A(K,J)*B(J)
```

In the preceding example, K is set to 1 and the inner DO is repeated, with J varying from 1 to 15. K is then set to 2, and the process is repeated; execution continues until K=16. Notice that both of the DO loops share the terminal statement 73.

6.4.6.1 Transfer of Control in Nested DO Loops

Within a nested DO loop, control can be transferred from an inner loop to an outer loop. However, a transfer from an outer loop to an inner loop is not permitted. For example, the following sequence is legal.

```
      DO 99 I=1,10
        .
        .
        .
      DO 40 M=1,30,2
        .
        .
        .
      GO TO 99
        .
        .
        .
40    D(M) = Q(M)
        .
        .
        .
99    CONTINUE
```

However, the following sequence is illegal, since control is being transferred from an outer loop to an inner loop.

```
      DO 99 I=1,10
        .
        .
        .
      GO TO 99
        .
        .
        .
      DO 99 J=1,15
        .
        .
        .
99    CONTINUE
```

The shared terminal statement is owned by the innermost DO.

6.5 DO Forever Statement

A DO forever statement specifies a DO loop having no explicit loop termination. Following execution of the associated terminal statement, provided this latter statement causes no transfer of control, control is unconditionally transferred to the first statement in the range of the DO forever statement.

The DO forever statement is designed to provide enhanced control structuring.

Syntax

```
DO [x] [,] [BEGIN]
```

x — The statement label of an executable statement following the DO forever statement and within the same program unit.

BEGIN — A keyword provided for the convenience of the programmer.

Rules for Use

- It is invalid to transfer control into the range of a DO loop from outside the loop.
- Within the range of a DO statement there may be other DO statements, in which case the DO statements are said to be nested.
- Exit from a DO forever loop occurs only upon execution of some control transfer statement within the loop range. The LEAVE statement is appropriate.

Example

```
DO
  .
  .
  .
IF (A.EQ.39.0) LEAVE
A = A + 1.0
END DO
SUM = A + B
```

6.6 DO UNTIL Statement

The DO UNTIL statement is an extension that provides enhanced control structuring.

Syntax

```
DO [x] [,] UNTIL (e)
```

x — The statement label of an executable statement following the DO UNTIL statement and within the same program unit.

e — A logical expression.

Execution of a DO UNTIL statement proceeds as follows:

- Control proceeds to the first executable statement in the DO range. Thus, the DO loop is executed at least once.
- Statements in the range of the DO UNTIL loop are executed in the same manner as a standard DO.
- Following the execution of the terminal statement of the loop, the expression e is evaluated. If e is .TRUE., the loop becomes inactive and control is transferred past the terminal statement in the same manner as a standard DO. If e is .FALSE., the execution process is repeated.

Note: Executing a statement within the loop range that explicitly transfers control outside the loop may also be used to exit a DO UNTIL loop.

Rules for Use

- It is invalid to transfer control into the range of a DO loop from outside the loop.
- Within the range of a DO statement there may be other DO statements, in which case the DO statements are said to be nested.

Example

```
DO UNTIL (A.EQ.33.0)
  .
  .
  .
A = A + 1.0
END DO
SUM = A + B
```

6.7 DO WHILE Statement

The DO WHILE statement is an extension that provides enhanced control structuring.

Syntax

```
DO [x] [,] WHILE (e)
```

x The statement label of an executable statement following the DO WHILE statement and within the same program unit.

e A logical expression.

Execution of a DO WHILE statement proceeds as follows:

1. The expression e is evaluated. If e is .FALSE., control is transferred past the associated terminal statement in the same manner as a standard DO. If e is .TRUE., control proceeds to the first executable statement in its range.

2. Statements in the range of the DO WHILE loop are executed in the same manner as a standard DO.

3. Following the execution of the terminal statement of the loop, the execution process is repeated with step 1 above.

Note: Executing a statement within the loop range that explicitly transfers control outside the loop may also be used to exit a DO WHILE loop.

Rules for Use

- It is invalid to transfer control into the range of a DO loop from a statement outside the loop.

- Within the range of a DO statement there may be other DO statements, in which case the DO statements are said to be nested.

Example

```
DO WHILE (A.LT.12.0)
  .
  .
  .
A = A + 1.0
END DO
ALL = A + B
```

6.8 LEAVE Statement

The LEAVE statement is an extension to the DO statements, including DO forever, DO UNTIL, and DO WHILE statements.

Syntax

```
LEAVE [x]
```

x — The statement label of the terminal statement of a DO loop (including a DO forever, DO UNTIL, or DO WHILE loop) statement.

Rules for Use

- The LEAVE statement may only occur within the range of the DO statement.
- If the statement label is present, execution of the LEAVE statement causes control to be transferred to the next executable statement following the terminal statement labeled x.
- If LEAVE is executed without a statement label, control is transferred from the innermost active DO loop only.

Examples

```
19      DO 33
          :
        A = A + 1.0
        IF (A.EQ.15.0) LEAVE 33
        SUM = A + B
33      CONTINUE
```

In the following example, control is transferred from the innermost active DO loop.

```
        DO 1 I=1,N
        DO 2 J=1,K
         :
        IF (J.EQ.I) LEAVE
2       CONTINUE
1       CONTINUE
```

Execution of LEAVE in the following example transfers control out of both loops.

```
        DO 10
        I = I+1
        DO 10
        J = J+1
        IF (J.EQ.4) LEAVE 10
10      CONTINUE
          :
```

6.9 CONTINUE Statement

The CONTINUE statement transfers control to the next executable statement. It is used primarily as the terminal statement of a DO loop when that loop would otherwise end illegally.

Syntax

```
CONTINUE
```

Example

In the following example, the CONTINUE statement is used to avoid ending the DO loop with an arithmetic IF statement.

```
        DO 15 K=1,10
          :
        IF (C2) 20,15,15
15      CONTINUE
          :
20      C2=C2 + .005
```

6.10 END DO Statement

The END DO statement is used as the terminal statement of a DO loop; it has no other effect.

Syntax

```
END DO
```

Rules for Use

- The END DO may only be used as the terminal statement of a DO loop.
- An END DO statement must be labeled if it terminates a DO specifying a label (see the first example below).
- An unlabelled END DO statement terminates only the DO loop that corresponds to the DO statement immediately preceding the END DO statement, i.e., an unlabelled END DO statement cannot terminate multiple DO statements.

Example

```
        DO 1 J=1,K
          :
1       END DO
        DO J=1,K
        DO I=1,N
          :
        END DO
        END DO
```

6.11 SELECT CASE Control Structure

The SELECT CASE control structure enables a user to select one block of statements for execution from several alternate blocks based upon the value of an expression.

The SELECT CASE construct,

```
      SELECT CASE
          CASE
block [   .
      [   .
      [   .
      END SELECT
```

contains:

- An initial SELECT CASE statement
- A set of statement blocks. (The first statement of each block is a CASE statement, a CASE DEFAULT statement, or an ELSE statement)
- A terminal END SELECT statement

There may be zero or more CASE blocks in a SELECT CASE construct. The execution of the SELECT CASE construct begins with the execution of the SELECT CASE statement. This causes control to be transferred to the CASE, CASE DEFAULT, or ELSE statement of one of the CASE blocks or to the END SELECT statement. If control is transferred to a CASE statement, the normal execution sequence is followed starting with that CASE statement. When the last executable statement of the CASE block is executed, control is transferred to the END SELECT statement.

Only an entire SELECT CASE construct may be nested within any block of statements. No transfer of control may occur into any CASE block from outside of the block.

6.11.1 SELECT CASE Statement

When SELECT CASE is executed, an expression is evaluated. The value of the expression determines the statement to which control is transferred.

Syntax

```
SELECT CASE exp
```

exp — An INTEGER*1, INTEGER*2, INTEGER*4, REAL*4, LOGICAL*1, or LOGICAL*4 expression.

Rules for Use

- If the value of exp equals none of the values specified in any of the CASE statements, and a CASE DEFAULT or ELSE statement is present in the SELECT CASE construct, control is transferred to the CASE DEFAULT or ELSE statement; if no CASE DEFAULT or ELSE statement is present, control is transferred to the END SELECT statement of the SELECT CASE construct.
- Transfer of control from outside the SELECT CASE structure to a statement within the structure is improper.
- The range of a case expression for an integer word is from -2147483648 to 2147483646.

6.11.2 CASE Statement

Execution of a CASE statement has no effect; it is used to mark the beginning of a CASE block and to specify the values that cause control to be transferred to a CASE block.

Syntax

CASE $const_1$ [,$const_2$] . . .

CASE DEFAULT

$const_i$ A constant expression or range of the form [(] l : u [)], where l and u are constant expressions.

Rules for Use

- The range forms of $const_i$ may be used with the integer or real data types.
- Note that CASE generates a full word value, which is compared to the SELECT CASE expression. For example, in comparing a byte I to a 1HH word, the right-justified, zero-filled word value of 1 is compared to the left-justified, blank-filled Hollerith constant 1HH or X'48202020' Therefore, the byte I must be compared to 1RH or X'00000048' to obtain correct results.
- The value of l must be less than the value of u.
- The form l : u specifies every value between the values of l and u including the endpoint values l and u, with the following exceptions:

  If the SELECT CASE type is REAL, ranges can be expressed in three additional forms:

  (l:u) indicating the range is from l to u excluding both l and u.

  l:u) indicating the range is from l to u including l, but excluding u

  (l:u indicating the range is from l to u excluding l, but including u.

- The type of $const_i$, l, and u must agree with the type of the SELECT CASE expression, exp.
- The CASE constant expression and ranges within a SELECT CASE structure must be distinct and nonoverlapping.
- At most, one CASE DEFAULT or ELSE statement may be specified in a SELECT CASE construct.

6.11.3 END SELECT Statement

Execution of END SELECT has no effect; it marks the end of the SELECT CASE construct.

Syntax

```
END SELECT
```

Example

```
SELECT CASE H+1.
CASE 1.
  .
  .
  .
CASE 2.:4.
  .
  .
  .
CASE DEFAULT
  .
  .
  .
CASE 5.:7.)       ! Range is from 5.0 up to but not including 7.0
  .
  .
  .
CASE (7.:12.      ! Range is up to 12.0 but excluding 7.0
  .
  .
  .
END SELECT
```

Depending upon the value of the SELECT CASE expression H+1., control is transferred to one of the case statements within the CASE block. If the value of the expression equals none of the values specified in any of the CASE statements, control is transferred to the CASE DEFAULT statement.

6.12 STOP Statement

Execution of the STOP statement causes the termination of a program.

Syntax

```
STOP n
```

n — A sequence of alphanumeric characters (the last eight nonblank characters are made available to the program's execution environment.)

Rules for Use

- If n appears, the message STOP, followed by the last eight nonblank alphanumeric characters of the string n, is output to the spooled printer output.
- There is no output if n does not appear.

6.13 END Statement

The END statement marks the end of a program unit. If control is transferred to the END statement in a main program, the effect is as if a STOP statement (with no character string) were encountered. If control is transferred to the END statement in a subprogram, the effect is as if a RETURN statement were encountered.

Syntax

```
END
```

Rules for Use

- Every program must end with this statement; the END statement must be the last (physical) source line of every program unit.
- If the END statement is the only executable statement found within a program MAIN, then no object code will be generated.
- No other statement in a program unit may have an initial line that appears to be an END statement.

Example

```
1 2 3 4 5 6 7 .....

      END                 Illegal
+     DO                  Continuation
                          Line

      END I               Legal
+       F                 Continuation
                          Line
```

6.14 PAUSE Statement

The PAUSE statement temporarily suspends program execution and displays a message on the operator's console or user terminal. When PAUSE is encountered during execution of an interactive program, the message PAUSE, followed by the last eight alphanumeric characters of the string n, is output to the operator's console or user terminal, depending on the mode of execution. Task execution is then suspended.

Syntax

```
PAUSE n
```

n — A sequence of alphanumeric characters (the last eight nonblank are displayed).

Rules for Use

- Execution of an interactive program may be resumed by the use of the CONTINUE or DEBUG statement. (Refer to the MPX-32 Reference Manual for a description of interactive processing).
- In batch or independent mode execution of the program after a PAUSE statement requires use of the OPCOM CONTINUE command.
- Execution continues with the first executable statement following the PAUSE statement if CONTINUE is used.

CHAPTER 7

SPECIFICATION STATEMENTS

7.1 Introduction

Specification statements are nonexecutable statements that specify the type and storage characteristics of variables, arrays, and functions. These statements must precede statement function statements, DATA statements, and any executable statements in a program unit.

The following specification statements are described in this chapter:

DIMENSION - provides information about storage required

Type Statements - override or confirm implicit typing

PARAMETER - names a constant

COMMON - provides information about storage

EQUIVALENCE - provides information about storage

EXTERNAL - identifies an external procedure

INTRINSIC - identifies an intrinsic function

SAVE - provides information about storage

Many of these statements involve the use of arrays, which are described in Chapter 3.

Three additional specification statements (EXTENDED BASE, EXTENDED BLOCK, EXTENDED DUMMY) are discussed in Chapter 13, Sections 13.1 through 13.4.

7.2 DIMENSION Statement

The DIMENSION statement specifies the number of dimensions in an array and the number of elements in each dimension.

A number of storage elements are allocated to each array named in the statement, and one storage element is assigned to each array element in each dimension.

Syntax

DIMENSION a_1(d[,d]. . .) [,a_i(d[,d]. . .)] . . .

a_i(d) An array declarator.

a_i The symbolic name of an array.

d A dimension declarator of the form [dl:]du where dl is the lower dimension bound (the default value is 1) and du is the upper bound. The dimension bounds can be expressions. The dimension size is du-dl+1.

Rules for Use

- The data type of an array determines the length of each storage element.
- The total number of storage elements assigned to an array is equal to the product of all dimension sizes in the array declarator for that array. For example,

  DIMENSION TABLE(4, 4), LES(5, 5, 10)

  defines array TABLE as having 16 elements and array LES as having 250 elements.
- An array can be dimensioned only once in a program unit.
- If a DIMENSION statement is used to declare an adjustable array and that array name is used in a SUBROUTINE, FUNCTION, or ENTRY statement, all variables in the array declarator for that array must either be passed to the SUBROUTINE, FUNCTION, or ENTRY statement, or the variables must be in COMMON or extended common. In addition, an array name and the actual parameters included in its declarator must be given in each call to the subroutine or function in which the array is referenced. For example, the following statement declares an adjustable array:

  DIMENSION DUM1 (J,K)

The name DUM1 must occur as a dummy argument in a SUBROUTINE, FUNCTION or ENTRY statement within the same program unit in which the array declarator occurs. Each of the dimension bounds J and K must either occur as a dummy argument in all of the SUBROUTINE, FUNCTION or ENTRY statements in which DUM1 occurs (in this case they occur in the SUBROUTINE statement for program unit SUB1), or as a variable name in a COMMON or EXTENDED BLOCK statement within the program unit.

```
PROGRAM ENTER
DIMENSION  AMTRX (5,6),  DMTRX (9, 9)
  :
CALL SUB1  (AMTRX, 5,6)
  :
CALL SUB1  (DMTRX, 3,3)
  :
END
SUBROUTINE  SUB1 (DUM1, J,K)
DIMENSION  DUM1 (J,K)
  :
END
```

7.3 Type Statements

In the absence of an IMPLICIT statement (including IMPLICIT NONE), variables, arrays, and functions (other than intrinsic functions) are automatically assigned the type INTEGER or REAL unless they are explicitly declared in type statements.

All variables, arrays, and functions (other than predefined functions) that are required to be DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, LOGICAL, CHARACTER, or integers of optional sizes must be implicitly or explicitly assigned the proper type in each program unit in which they are used.

The following table presents lengths (in bytes) for specified types:

Table 7-1
Standard and Optional Type Lengths

Type	Standard Length (bytes)	Optional Length (bytes)
INTEGER	4	1, 2, or 8
REAL	4	8
DOUBLE PRECISION	8	None
COMPLEX	8	16
LOGICAL	4	1
BIT	one bit	None
CHARACTER	1	1 through 4095

7.3.1 Explicit Type Statement

Type statements explicitly define the types of variables, arrays, or functions. Optionally, they can declare arrays and define initial values for variables or arrays.

Syntax

type [*s] a_1 [*z_1] [d_1] [*z_1] [/x_1/],. . .,a_n [*z_n] [d_n] [*z_n] [/x_n/]

type — Integer, real, double precision, complex, logical, bit, or character. CHAR can be used as a synonym for INTEGER*1.

s — One of the permissible length specifications for the associated type (see Table 7-1). (FORTRAN 77+ implementation, not ANSI standard.)

a_1,...,a_n — Variable, array, or function names.

d_1,...,d_n — Dimension declarators of the form (d[,d]. . .). Each d may have the form [dl:] du, where dl is the lower bound (the default value is 1 if dl is absent) and du is the upper bound.

z_1,...,z_n — Length specifications that apply only to the name immediately preceding the specification. (FORTRAN 77+ implementation, not ANSI standard.)

x_1,...,x_n — Constants or lists of constants (in the case of an array specification) that represent initial data values. Unlike DATA statement constants, these constants do not correspond to a list of variables or arrays, but to the preceding variable or array. Refer to the DATA statement for a description of the permissible types of constants. (FORTRAN 77+ implementation, not ANSI standard.)

Rules for Use

- Explicit type specifications override any implicit type specifications.
- Length specifiers s and z must not be used concurrently to specify the same array, variable, or function. For example, the following statements are correct:

```
INTEGER HW*2 (10)          CHARACTER C*100 (10)
or                         or
INTEGER HW (10)*2          CHARACTER C (10)*100
```

However, the following statements are incorrect:

```
INTEGER HW*2 (10)*2
CHARACTER C*100 (10)*100
```

The name DUM1 must occur as a dummy argument in a SUBROUTINE, FUNCTION or ENTRY statement within the same program unit in which the array declarator occurs. Each of the dimension bounds J and K must either occur as a dummy argument in all of the SUBROUTINE, FUNCTION or ENTRY statements in which DUM1 occurs (in this case they occur in the SUBROUTINE statement for program unit SUB1), or as a variable name in a COMMON or EXTENDED BLOCK statement within the program unit.

```
PROGRAM ENTER
DIMENSION  AMTRX (5,6),  DMTRX (9, 9)
 :
CALL SUB1  (AMTRX, 5,6)
 :
CALL SUB1  (DMTRX, 3,3)
 :
END
SUBROUTINE  SUB1 (DUM1, J,K)
DIMENSION  DUM1 (J,K)
 :
END
```

7.3 Type Statements

In the absence of an IMPLICIT statement (including IMPLICIT NONE), variables, arrays, and functions (other than intrinsic functions) are automatically assigned the type INTEGER or REAL unless they are explicitly declared in type statements.

All variables, arrays, and functions (other than predefined functions) that are required to be DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, LOGICAL, CHARACTER, or integers of optional sizes must be implicitly or explicitly assigned the proper type in each program unit in which they are used.

The following table presents lengths (in bytes) for specified types:

Table 7-1
Standard and Optional Type Lengths

Type	Standard Length (bytes)	Optional Length (bytes)
INTEGER	4	1, 2, or 8
REAL	4	8
DOUBLE PRECISION	8	None
COMPLEX	8	16
LOGICAL	4	1
BIT	one bit	None
CHARACTER	1	1 through 4095

7.3.1 Explicit Type Statement

Type statements explicitly define the types of variables, arrays, or functions. Optionally, they can declare arrays and define initial values for variables or arrays.

Syntax

type [*s] a_1 [*z_1] [d_1] [*z_1] [/x_1/],. . .,a_n [*z_n] [d_n] [*z_n] [/x_n/]

type — Integer, real, double precision, complex, logical, bit, or character. CHAR can be used as a synonym for INTEGER*1.

s — One of the permissible length specifications for the associated type (see Table 7-1). (FORTRAN 77+ implementation, not ANSI standard.)

a_1,...,a_n — Variable, array, or function names.

d_1,...,d_n — Dimension declarators of the form (d[,d]. . .). Each d may have the form [dl:] du, where dl is the lower bound (the default value is 1 if dl is absent) and du is the upper bound.

z_1,...,z_n — Length specifications that apply only to the name immediately preceding the specification. (FORTRAN 77+ implementation, not ANSI standard.)

x_1,...,x_n — Constants or lists of constants (in the case of an array specification) that represent initial data values. Unlike DATA statement constants, these constants do not correspond to a list of variables or arrays, but to the preceding variable or array. Refer to the DATA statement for a description of the permissible types of constants. (FORTRAN 77+ implementation, not ANSI standard.)

Rules for Use

- Explicit type specifications override any implicit type specifications.
- Length specifiers s and z must not be used concurrently to specify the same array, variable, or function. For example, the following statements are correct:

```
INTEGER HW*2 (10)          CHARACTER C*100 (10)
or                         or
INTEGER HW (10)*2          CHARACTER C (10)*100
```

However, the following statements are incorrect:

```
INTEGER HW*2 (10)*2
CHARACTER C*100 (10)*100
```

- The type statement must precede any reference to variables, arrays, or functions that it defines except in the case of FUNCTION, SUBROUTINE, ENTRY, or NAMELIST statements. For example, the statements

```
PARAMETER (I=10)
INTEGER I
  .
  .
  .
```

would result in an error; however, the following is correct

```
NAMELIST /NLIST/I,J,K
INTEGER I
REAL J
COMPLEX K
  .
  .
  .
```

Examples

```
REAL BX, ITEA, KEPH
```

BX, ITEA, and KEPH are of type real. Note that BX is redundantly typed since it would be of type real by default.

```
INTEGER*1 ED
```

ED is defined as an integer byte.

```
BIT STATE
```

STATE is defined as a bit.

```
REAL*8 SUM/3.5D2/
```

SUM is defined as double precision (REAL*8 is equivalent to double precision). The variable is initialized to the value 350.0 converted to internal floating point doubleword form.

```
REAL LOT, DBL*8, IT*4
```

LOT and IT are defined as real, while DBL is defined as double precision.

```
INTEGER JMP*8 (15), RATE*2 (4) /5, 10, 12, -3/
```

JMP*8 (15) is a constant array declarator specifying a one-dimensional array of 15 integer doublewords. RATE*2 (4) is a one-dimensional array of four integer halfwords.

Each element of the array is initialized as follows:

RATE (1) has the value 5
RATE (2) has the value 10
RATE (3) has the value 12
RATE (4) has the value -3

In the following example, L1 and LTEMP are defined to be logical words. In addition, LTEMP is initialized as .TRUE.

```
LOGICAL L1, LTEMP/.TRUE./
```

7.3.2 Explicit CHARACTER Statement

As in numeric type declaration statements, the explicit CHARACTER type declaration statement explicitly defines the data types of specified symbolic names.

Syntax

```
CHARACTER [*s [,] ]name [,name]...
```

s — The length (number of characters) of a character variable, character array element, character function, or character constant that has a symbolic name. s is referred to as the length specification and is one of the following:

- An unsigned, nonzero, integer constant.
- A positive value integer constant expression enclosed in parentheses.
- An asterisk enclosed in parentheses (*). This form may be used for the following:
  - dummy arguments in subroutines or functions
  - symbolic constants defined in PARAMETER statements in which the assumed length is the same as the defining character constant
  - names of functions in FUNCTION subprograms
  - entry names in FUNCTION subprograms
  - statement function names
  - internal function names

name — A variable name, array name, symbolic name of a constant, or function name that can have one of the following forms:

```
v [*s]
a [(d)] [*s]
a [*s] [(d)]
```

v — A variable name, symbolic name of a constant, or function name.

a — An array name.

a(d) — An array declarator.

Rules for Use

- A length immediately following the word CHARACTER is the length specification for each entity in the statement that does not have its own length specification. Thus, in the following statement, a length of 6 applies to NAME and LAST.

```
CHARACTER*6  NAME(10), FIRST*8, LAST
```

- The maximum number of characters, s, allowed in any explicit CHARACTER statement is 4095.

- A length specification immediately following an entity is the length specification for only that entity (as in FIRST*8 in the preceding example).

- A length specification for an array refers to each array element. For example, the length 6 applies to each element of array NAME in the preceding example.

- The default for s is one. Therefore, the statement

```
CHARACTER  LETTER(26)
```

  specifies an array, LETTER, containing 26 one-character elements.

- A dummy argument with a declared length of (*) assumes the length of the associated actual argument for each reference to the subroutine or function. If the associated actual argument is an array name, the length assumed by the dummy argument is the length of each array element in the actual argument array.

```
CHARACTER*10 A
CHARACTER*20 B
CALL SUB(A)
CALL SUB(B)
  .
  .
  .
SUBROUTINE SUB(X)
CHARACTER X*(*)
DO 100  I = 1, LEN(X)
```

- If an external function has a length of (*) declared in a function subprogram, a FUNCTION or ENTRY statement in the same subprogram must reference the function name. When a reference to such a function is executed, the function assumes the length specified in the calling program unit.

- A character constant (denoted by a symbolic name) having a declared length of (*) assumes the length of its corresponding constant expression in a PARAMETER statement.

- Because the data type CHAR (synonymous with INTEGER*1) is supported under FORTRAN 77+, do not use statements such as:

```
CHAR     ACTER*4     ALPHA
```

7.3.3 IMPLICIT Statement

The IMPLICIT statement changes or confirms the default integer or real typing. It also defines the length (in bytes) of the items implicitly typed by the IMPLICIT statement.

Syntax

IMPLICIT type [*s](a_1[,a_2,. . .,a_n]),. . .[,type[*s] (a_1 [,a_2,. . .,a_n])]

type	Integer, real, double precision, complex, logical, bit, or character. (CHAR can be used as a synonym for INTEGER*1.)
s	An unsigned integer that is one of the permissible length specifications for its associated type (refer to section 7.3 for permissible length specifications).
a_1,a_2,. . .,a_n	Single alphabetic characters or a range of characters in alphabetic sequence. The range of alphabetic characters is denoted by a minus sign between the first and last characters of the range; for example, A-G.

Rules for Use

- In a subprogram, the IMPLICIT specification statement affects the types of dummy arguments as well as the types of subsequently mentioned symbolic names. However, the name of the subprogram is not affected.
- An IMPLICIT statement cannot type the same letter differently; therefore, the following example is invalid and will result in an error:

```
IMPLICIT  INTEGER (A-F), REAL(D)
```

- The names of intrinsic functions are not affected by the use of the IMPLICIT statement.
- Explicit data type specifications override the IMPLICIT typing mechanism.

Examples

```
IMPLICIT  INTEGER (B),  INTEGER*2 (J-N),  COMPLEX (G,P)
```

All variables, arrays, or functions (other than intrinsic functions) that have names beginning with B are of type INTEGER (INTEGER*4); those that have names beginning with J, K, L, M, or N are of type INTEGER*2; those that have names beginning with G or P are of type COMPLEX (COMPLEX*8).

```
   IMPLICIT  REAL*8 (A-C),  DOUBLE PRECISION (X),
+  LOGICAL (T), COMPLEX*16 (D)
```

All variables, arrays, or functions (other than intrinsic functions) that have names beginning with A, B, C, or X are of type DOUBLE PRECISION (REAL*8); those that have names beginning with T are of type LOGICAL (LOGICAL*4); those that have names beginning with D are of type COMPLEX*16.

```
IMPLICIT CHARACTER (Q, T), CHARACTER*8 (F)
```

All variables, arrays, or functions (other than intrinsic functions) that have names beginning with Q or T are of type CHARACTER (a length of one is assigned by default); those that have names beginning with F are of type CHARACTER and are eight characters long.

7.3.4 IMPLICIT NONE Statement

The IMPLICIT NONE statement disables implicit typing. All variable names, array names, and function names (except the names of intrinsic functions) must be explicitly typed. However, if an IMPLICIT NONE statement is used with a function declared external, the function type is implicitly INTEGER unless otherwise explicitly typed.

Syntax

```
IMPLICIT NONE
```

Rule for Use

A program unit containing an IMPLICIT NONE statement cannot contain any other IMPLICIT statement.

Example

```
IMPLICIT NONE
```

The following sequence will cause a compiler diagnostic indicating that I has not been typed:

```
IMPLICIT NONE
I = 0
```

7.4 PARAMETER Statement

The PARAMETER statement assigns a symbolic name to a constant. Once defined, a symbolic name can appear in any subsequent statement within the program unit as a primary in an expression, or it can appear in a DATA statement.

Syntax

PARAMETER $(s_1 = e_1, \ldots, s_i = e_i)$

s_i Symbolic names.

e_i Constant expressions.

Rules for Use

- If the symbolic name s is of type integer, real, double precision, or complex, the corresponding expression e must be an arithmetic constant expression. If the symbolic name is of type character or logical, the corresponding expression must be a character constant expression or a logical constant expression, respectively.
- The value of s is determined from the expression e in accordance with the rules for assignment statements. Character relational expressions are not supported in the PARAMETER statement.
- Any symbolic name of a constant in an expression must have been defined previously in a PARAMETER statement within the same program unit.
- If a constant's symbolic name and length are not of the default type, they must be specified by a type statement or an IMPLICIT statement before the name can be used in a PARAMETER statement. The constant's length must not be changed by subsequent statements.
- A symbolic name of a constant must not be part of a format specification.
- A symbolic name of a constant must not form part of another constant, for example, any part of a complex constant.
- A symbolic name in a PARAMETER statement can be used only to identify the corresponding constant in that program unit.
- When a parameter is declared as INTEGER*1, the internal constant generated by the compiler for this variable will be a halfword constant.

Examples

```
PARAMETER (INT = 58, BETA = 1.10)
LOGICAL STOW
PARAMETER (STOW = .TRUE.)
```

7.5 COMMON Statement

The COMMON statement defines one or more contiguous areas (blocks) of storage. A symbolic name identifies each block; however, the symbolic name for one block in a program unit can be omitted. This block is the blank common block. COMMON statements also define the order of variables and arrays in each common block.

Syntax

COMMON [/ y_1 /] a_1 [[,]/y_2/a_2] ...

y_i Symbolic names, called block names. If the first y_i is blank, the first pair of slashes can be omitted (the block of storage so indicated is called blank common).

a_i Lists of variable names, array names, or constant array declarators separated by commas. (The elements in a_i compose the common block storage area specified by the name y_i.)

Rules for Use

- Entities within a common block should be either all numeric data or all character data to comply with ANSI X3.9-1978; however, a common block can contain both numeric and character data in FORTRAN 77+.
- There can be at most one blank common block in an executable program, but there can be several named common blocks.
- A DATA statement cannot be used to initialize a variable or array in blank common storage or in a named common block with one exception: A DATA statement can initialize data in a named common block when used in a BLOCK DATA subprogram.
- A common block name can appear more than once in the same COMMON statement or in more than one COMMON statement.
- The size of a common area can be increased by the use of EQUIVALENCE statements; however, an EQUIVALENCE statement must not cause the association of two different common block storage sequences that are in the same program unit.
- Named common blocks are independent of each other and of blank common.
- The length of a common block is the number of storage units required to contain the variables and arrays declared in the COMMON statement (or statements), unless it is expanded by the use of EQUIVALENCE statements or as a requirement to meet bounding conditions. The FORTRAN 77+ compiler does not validate or enforce consistent common block lengths. For consistency and control, it is good practice to have common block definitions exactly the same in all program units. Operating system loaders may have different rules for processing common blocks of different lengths, defined in different program units.

- A mixture of data types in a common block often requires unused space to be generated within the common block. In the following program for example, unused space is generated in storage locations 1, 4, 5, 6, and 7. A common block, when loaded, is aligned at a file boundary (a file boundary occurs at every eighth word).

```
COMMON/C/IBYTE,IHALFWORD,IDBLWORD
INTEGER*1 IBYTE
INTEGER*2 IHALFWORD
INTEGER*8 IDBLWORD
    .
    .
    .
```

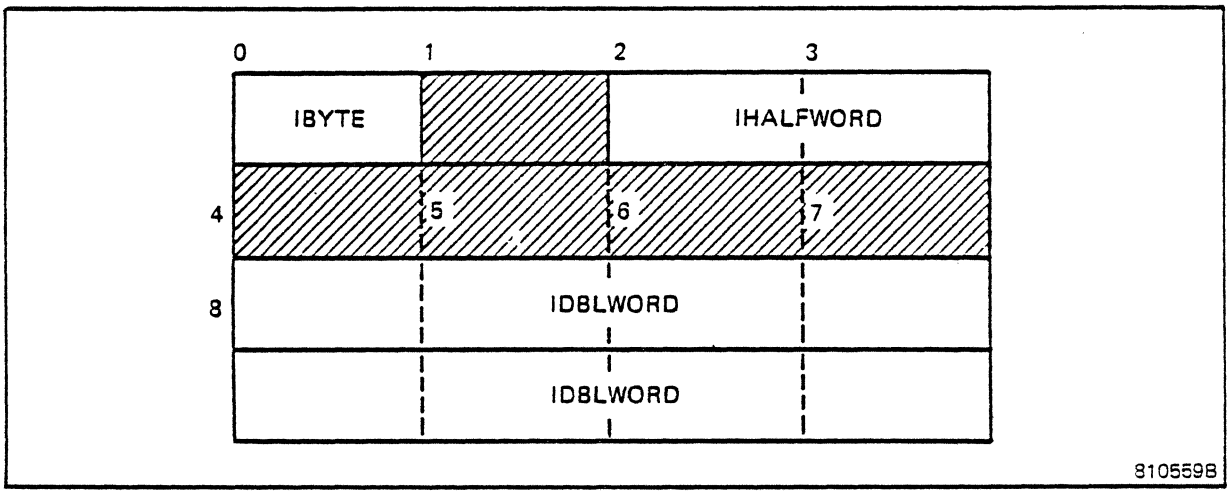


Figure 7-1. Mixed Data Type Storage in Common

- To avoid problems with unused space in common blocks, data items can be arranged in descending order of boundary alignment requirements; e.g., DOUBLE PRECISION (REAL*8) before REAL (REAL*4), before INTEGER*2. In this way, no extra space is needed to produce boundary alignment.

- The maximum common block size depends on the hardware, the use of extended memory and the combined size of the operating system and task. The operating system and task (including non-extended commons) cannot exceed 128 KW. (For maximum sizes of extended memory, see Chapter 13, Extended Addressing.)

Examples

```
COMMON // A1, B1 /CDATA/ ZOT(3,3) // T2,Z3
```

In the preceding example, A1, B1, T2, and Z3 are assigned to blank common in that order. Note that the first pair of slashes could have been omitted. CDATA names common block storage for the nine-element array ZOT. The array must not have been previously dimensioned. T2 and Z3 are assigned to blank common after B1.

Main Program

```
COMMON HEAT, X/BLK1/KILO, Q
   .
   .
   .
CALL FIGURE
   .
   .
   .
```

Subprogram

```
SUBROUTINE FIGURE
COMMON /BLK1/MR, T/ /ALF, BT
   .
   .
   .
RETURN
END
```

The COMMON statement in the main program puts HEAT and X in the blank common block and puts KILO and Q in a named common block BLK1. The COMMON statement in the subroutine makes ALF and BT correspond to HEAT and X in the blank common block and makes MR and T correspond to KILO and Q in BLK1.

To avoid confusion, use the same names in the main program and subprogram, as shown in the following example:

Main Program

```
COMMON HEAT, X/BLK1/KILO, Q
   .
   .
   .
CALL FIGURE
   .
   .
   .
```

Subprogram

```
SUBROUTINE FIGURE
COMMON HEAT, X/BLK1/KILO, Q
   .
   .
   .
RETURN
END
```

7.5.1 Global Common

A general description of GLOBAL COMMON can be found in the MPX-32 Reference Manual.

The range of virtual addresses occupied by a GLOBAL COMMON partition must be defined by declaring it as a static core partition at system generation (SYSGEN) time or as a dynamic partition by the volume manager CREATE COMMON directive before a user can execute a FORTRAN program containing a GLOBAL COMMON. If a partition is dynamic, access to it must be established by executing the X:INCL (Compatible mode) or

the X_INCLD (Native mode) service. For larger commons, extended addressing must be used. The GLOBAL COMMON may be as large as extended memory, i.e., .5MB on a 32/77, 1.5MB on a 32/27 or 32/87, and 15.5MB (minus the operating system) on a 32/67 and 32/97. Designation of a GLOBAL COMMON is made by a FORTRAN COMMON statement:

Syntax

COMMON / GLOBAL## / $a_1,a_2,\ldots a_i$

A pair of decimal digits from 00 to 99.

a_i Variable names, array names, or constant array declarators.

Example

```
COMMON/GLOBAL11/A,B,C
```

In this example, space for the three variables A, B, and C is allocated within the GLOBAL COMMON area GLOBAL11.

7.5.2 Datapool

DATAPOOL, a resident memory partition that serves as a special common block, is referenced by the COMMON statement. However, DATAPOOL items are bound to their addresses by reference to a DATAPOOL dictionary at catalog time, rather than at compile time. The DATAPOOL partition must be established as a static core partition at SYSGEN time or as a dynamic partition using the file manager or volume manager utilities. If the partition is dynamic, access to it must be established by calling the X:INCL (Compatible mode) or the X_INCLD (Native mode) service. However, if the DATAPOOL partition is to lie in the extended address space of the task, the X_DPXMNT or X_INCXDP services must be called. Note: These services are only available in native mode. Datapools cannot be in extended memory in compatible mode.

Additional information on DATAPOOL can be found in the MPX-32 Reference Manual. Information on the services mentioned above can be found in the Scientific Run-Time Library Reference Manual.

If multiple datapool areas are needed, there are 100 reserved common names that can be referenced as datapools. The names are DPOOL00, DPOOL01, through DPOOL99. Up to 16 of the datapools (DATAPOOL or DPOOL00 - DPOOL99) can be used in the same task. These DPOOL partitions are accessed in the same way as DATAPOOL. If a datapool is in extended memory, X_INCXDP is used to include the partition. X_INCXDP can also be used for non-extended datapools. For more information, refer to the Scientific Runtime Library Reference Manual.

Syntax

COMMON / { DATAPOOL / DPOOL## } / $a_1,a_2,\ldots,a_i$

a_i Lists of variable names, array names, or constant array declarators.

A pair of decimal digits from 00 to 99.

Example

```
COMMON/DATAPOOL/A,B,C
```

In this example, the variables A, B, and C must have been entered in a DATAPOOL dictionary. Their relative addresses within the DATAPOOL partition must be found in this dictionary. Access to a DATAPOOL is acquired in the same manner as access to a GLOBAL COMMON partition.

Rules for Use

- DATAPOOL list items may be in any order. The same execution would result if the previous example had been C, B, A. However, maintain compatible typing and sizing.
- DATAPOOL may be a shared area, and a number of users can reference it at the same time. Avoid establishing conflicting DATAPOOL dictionaries, although it is valid to do so.
- GLOBAL COMMON and DATAPOOL cannot be initialized by a BLOCK DATA subprogram.
- If the DATAPOOL partition is included in the user's extended address space, the EXTENDED BLOCK statement must be used to establish the base address for the DATAPOOL. For example:

```
EXTENDED BLOCK/DATAPOOL/A,B
    .
    .
    .
```

7.6 EQUIVALENCE Statement

The EQUIVALENCE statement permits two or more entities to share the same storage unit(s). The statement specifies that the storage sequence of the entities appearing in nlist have the same first storage unit. Such equivalence implies storage sharing only, not mathematical equivalence.

Syntax

```
EQUIVALENCE (nlist) [, (nlist)]
```

nlist — A list of variables, arrays, array elements, and character substring references, separated by commas.

Rules for Use

- An entity of type CHARACTER must be specified in the EQUIVALENCE statement only with other entities of type CHARACTER in order to comply with ANSI standards (X3.9-1978) and to avoid problems in transportability. FORTRAN 77+ does allow equivalence between numeric and character entities.

Examples

```
DOUBLE PRECISION DMOD
INTEGER  IABOR(2)
EQUIVALENCE (DMOD, IABOR(1))
```

The EQUIVALENCE statement in the preceding example causes the two elements of integer array IABOR to occupy the same storage as the double precision variable DMOD.

```
CHARACTER  DAY*16, STRM*10
EQUIVALENCE (DAY,STRM)
```

The EQUIVALENCE statement in this example causes the first characters of the character variables DAY and STRM to share the same storage location. The character variable STRM is equivalent to the substring DAY(1:10).

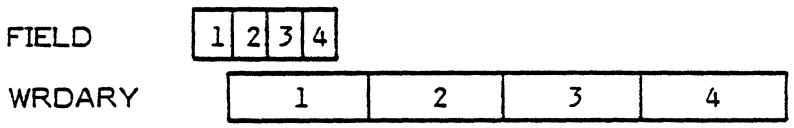
7.6.1 EQUIVALENCE and Boundaries

The boundary alignment of each element within an equivalence group is consistent, if possible, with its type:

- Complex, double precision elements must begin on a half-file (four-word) boundary.
- Double precision, complex, and integer doubleword variables or arrays must begin on a doubleword boundary.
- Integer word, real, and logical word variables or arrays must begin on a word boundary.
- Integer halfword variables or arrays must begin on a halfword boundary.
- Integer byte and logical byte variables or arrays begin on a byte boundary.
- Logical bit variables or arrays align at any bit.
- Character variables will start on arbitrary byte boundaries.

For example, the following EQUIVALENCE statement forces the byte element FIELD(2) to be on a word boundary.

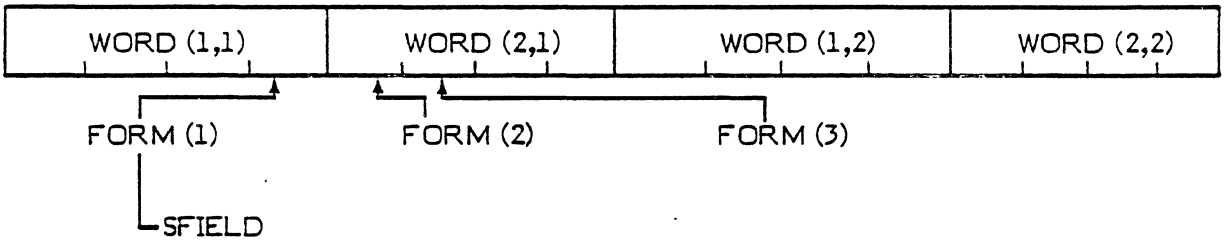
```
INTEGER  FIELD*1 (4),  WRDARY*4 (4)
EQUIVALENCE (WRDARY (1),  FIELD (2))
```



Care must be taken to insure that no prior constraint has been placed on entities that appear in the same EQUIVALENCE statement; i.e., entities that are in common or entities that have appeared in a previous EQUIVALENCE statement.

The following EQUIVALENCE statement results in an error:

```
INTEGER SFIELD*2, WORD*4 (2,2), FORM*1 (3)
EQUIVALENCE (FORM (2), WORD (2,1)), (SFIELD, FORM(1))
```



Note that FORM (2) aligns with byte 1 of WORD (2,1). Thus, FORM (1) aligns with byte 4 of WORD (1,1) and FORM (3) aligns with byte 2 of WORD (2,1). Due to this alignment, SFIELD attempts to align on FORM (1); i.e., byte 4 of WORD (1,1), which results in an error because SFIELD is declared an integer halfword and must begin on a halfword boundary.

Variables of different data types can be made equivalent by making components of a lower-ranked data type equivalent to a single component of a higher-ranked data type. For example, an integer variable can be made equivalent to a complex variable (the integer variable shares storage with the real part of the complex variable).

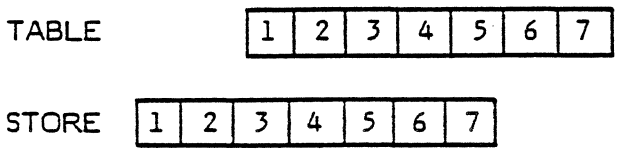
7.6.2 EQUIVALENCE and Arrays

When an element of one array is made equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between the corresponding elements of the two arrays. Thus, if the first elements of two arrays of equal size are made equivalent, both arrays share the same storage space. For example, the arrays in the following example would share the same storage space as a result of the EQUIVALENCE statement.

```
INTEGER STORE(7), TABLE(7)
EQUIVALENCE (STORE,TABLE)
```

If the third element of array STORE is made equivalent to the first element of array TABLE, the last five elements of STORE overlay the first five elements of array TABLE.

```
EQUIVALENCE (STORE(3), TABLE (1))
```



The following EQUIVALENCE statement results in the array storage indicated in Table 7-2.

```
DIMENSION  STORE(2,2), TABLE(2,2,2)
EQUIVALENCE (STORE(2,2), TABLE(1,2,2))
```

The following statements also align the two arrays as shown in Table 7-2.

```
EQUIVALENCE (STORE, TABLE(2,2,1))
EQUIVALENCE (TABLE(1,1,2), STORE(2,1))
```

EQUIVALENCE statements must not be used to assign the same storage locations to two or more elements of the same array. For example, the following statement results in an error:

```
EQUIVALENCE (XTABLE(6), XTABLE(15))
```

In addition, memory locations must not be assigned in a way that is inconsistent with the normal linear storage of array elements. For example, the first element of one array cannot be made equivalent to the first element of another array and then the second element of the first array made equivalent with the sixth element of the other array.

```
EQUIVALENCE (XTABLE(1), YSTORE(1))
      .
      .
      .
EQUIVALENCE (XTABLE(2), YSTORE(6))
```

Table 7-2
Equivalence of Array Storage

ARRAY TABLE		ARRAY STORE	
Array Element	Element Number	Array Element	Element Number
TABLE (1,1,1)	1		
TABLE (2,1,1)	2		
TABLE (1,2,1)	3		
TABLE (2,2,1)	4	STORE (1,1)	1
TABLE (1,1,2)	5	STORE (2,1)	2
TABLE (2,1,2)	6	STORE (1,2)	3
TABLE (1,2,2)	7	STORE (2,2)	4
TABLE (2,2,2)	8		

810560A

If an EQUIVALENCE statement contains an array element, the number of subscripts must be the same as the number of dimensions established by the array declarator, or it must be a single subscript that specifies the array element number relative to the first element of the array. For example, if the dimensionality of an array Z has been declared as Z(3,3), within different EQUIVALENCE statements the terms Z(6) and Z(3,2) have the same meaning (the sixth element and element (3,2) are the same).

Arrays can be made equivalent with nonunity lower bounds. For example, an array defined as A(2:3,4) is a sequence of eight storage locations. A reference to A(2,2) refers to the third element in the sequence. The following statement can be used to make array A(2:3,4) share the same storage with array B(2:4,4):

```
EQUIVALENCE (A(3,4), B(2,4))
```

The entire array A shares part of the storage space allocated to array B. Figure 7-3 shows how these statements align the arrays.

Note that the following statements also align the arrays as shown in Table 7-3:

```
EQUIVALENCE (A, B(4, 1))
EQUIVALENCE (B(3, 2), A(2, 2))
```

Table 7-3
Equivalence of Arrays with Nonunity Lower Bounds

ARRAY A		ARRAY B	
Array Element	Element Number	Array Element	Element Number
		B(2,1)	1
		B(3,1)	2
A(2,1)	1	B(4,1)	3
A(3,1)	2	B(2,2)	4
A(2,2)	3	B(3,2)	5
A(3,2)	4	B(4,2)	6
A(2,3)	5	B(2,3)	7
A(3,3)	6	B(3,3)	8
A(2,4)	7	B(4,3)	9
A(3,4)	8	B(2,4)	10
		B(3,4)	11
		B(4,4)	12

810561A

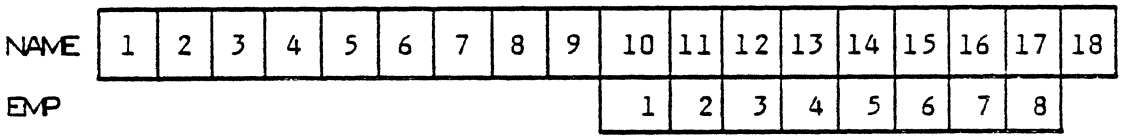
7.6.3 EQUIVALENCE and Substrings

When a character substring is made equivalent to another character substring, the EQUIVALENCE statement also sets equivalences between the other corresponding characters in the character entities.

For example,

```
CHARACTER  NAME*18,  EMP*8
EQUIVALENCE  (NAME (11:13), EMP (2:4))
```

As a result of the preceding statements, the character variables NAME and EMP share space as illustrated below:



The following statement also aligns the arrays as shown in the preceding example:

```
EQUIVALENCE  (NAME (10:10), EMP (1:1))
```

If the character substring references are array elements, the EQUIVALENCE statement sets equivalences between the other corresponding characters in the complete arrays (refer to Figure 7-2).

Character elements of arrays can overlap at any character position. For example, the statements

```
CHARACTER  TABLE(50)*2,  TOOL(5)*5
EQUIVALENCE (TABLE(3)(1:2),  TOOL(2)(3:4))
```

result in overlapping character positions, as indicated in Figure 7-2.

The EQUIVALENCE statement cannot be used to assign the same storage location to two or more substrings that start at different positions in the same character variable or character array. For example, the following EQUIVALENCE statement will cause an error condition:

```
CHARACTER NAME*10, ID*9
EQUIVALENCE (NAME(10:13), ID(2:5), NAME(1:3))
```

In addition, the EQUIVALENCE statement cannot be used to assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays.

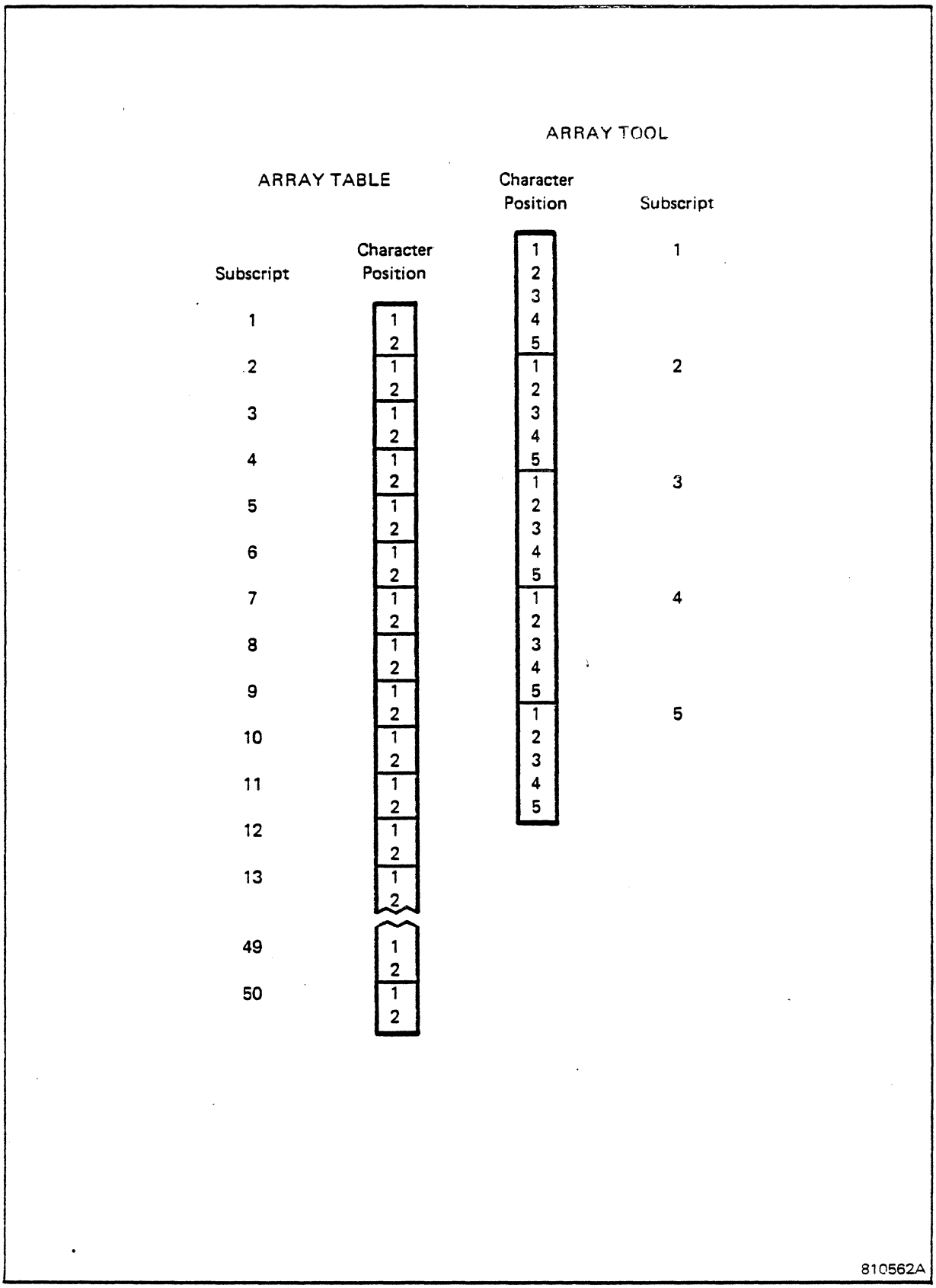


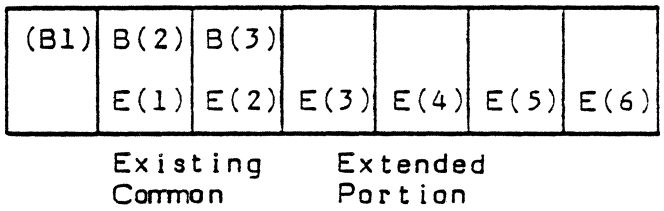
Figure 7-2. Equivalence of Character Arrays

7.6.4 EQUIVALENCE and COMMON Interaction

Variables may be made equivalent to variables in a common block through the use of the EQUIVALENCE statement.

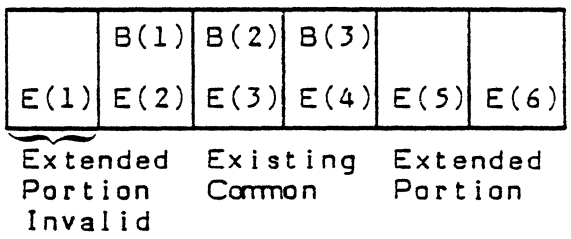
When components are made equivalent to entities stored in a common block, the common block can be extended beyond its original boundaries. However, it can only be extended beyond the last element of the previously established common block. The common block cannot be extended in such a way as to place the extended portion before the first element of the existing common block. For example, the following extension is valid:

```
DIMENSION B(3),E(6)
COMMON B
EQUIVALENCE (B(2),E(1))
```



The following extension is invalid since the common block would be extended before the first element of the existing common block.

```
DIMENSION     B(3),E(6)
COMMON B
EQUIVALENCE (B(2),E(3))
```



If two components are assigned to common blocks, equivalence cannot be established between them through the use of the EQUIVALENCE statement. The following example is invalid:

```
DIMENSION D(5),XTABLE(20),B(10), A(10)
COMMON A,B
EQUIVALENCE (A(7),B(3),XTABLE(6))
```

Care must be taken when using an EQUIVALENCE statement with an array that is in common with another, larger array that is not in common. For example,

```
DIMENSION T(7)
COMMON A(3), B(4)
```

stores arrays A and B in sequence as indicated below:

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

The following statement causes the entire common storage to be made equivalent:

```
EQUIVALENCE (A(1), T(1))
```

T(1)	T(2)	T(3)	T(4)	T(5)	T(6)	T(7)
A(1)	A(2)	A(3)	B(1)	B(2)	B(3)	B(4)

If the value of T(7) is now changed (as in T(7) = X), the value of B(4) is also changed.

A common block begins on an eight-word boundary. The boundary alignment of an entry in a common block cannot be changed by equivalence. For example, the following statements are inconsistent and will result in an error:

```
COMMON X,Y,Z
DOUBLE PRECISION D
EQUIVALENCE (D,Y)
```

7.7 EXTERNAL Statement

The EXTERNAL statement identifies a symbolic name as representing an external procedure, a dummy procedure, or a block data subprogram. An external procedure name or a dummy procedure name so declared can then appear as an actual argument to a subprogram (refer to Chapter 9). In an EXTERNAL statement, the occurrence of a block data subprogram name enables the block data subprogram to be loaded by name from a binary object file or library (refer to Chapter 10).

The EXTERNAL statement also indicates that user-defined functions are to be used rather than specific intrinsic functions having the same name. An EXTERNAL statement containing the names of all external procedures used by a program unit insures that those procedures will not be interpreted as intrinsic functions by the compiler. Thus, the function name will not have the predetermined type of the intrinsic function of the same name, and no inline code generation will occur as for certain intrinsic functions. However, it is the user's responsibility to provide an external user-defined procedure at catalog time for those function names that, as intrinsic function names, correspond to library procedures called as external procedures.

Syntax

```
EXTERNAL proc [,proc]. . .
```

proc — The name of an external procedure, dummy procedure, or block data subprogram.

Rules for Use

- An EXTERNAL statement must precede statement functions and executable statements.
- An external subprogram name used as an argument in a subprogram reference must have appeared in a preceding EXTERNAL statement.

- A function name must not appear in both an EXTERNAL and an INTRINSIC statement within a program unit.
- A symbolic name may represent an external procedure in the absence of the EXTERNAL statement.

Examples

Main Program

```
EXTERNAL SUM, AFUNC, BD
   .
   .
   .
CALL SUBR (SUM, AFUNC, X, Y)
```

Subprograms

```
SUBROUTINE SUM (A, B, C)
  A = B+C
RETURN
END

BLOCK DATA BD
   .
   .
   .
END

FUNCTION AFUNC (X,Y)
  AFUNC = X*Y
RETURN
END
```

In the preceding main program, SUM and AFUNC are subprogram names used as arguments in the subroutine SUBR. The appearance of BD in the EXTERNAL statement assures that the block data program BD is loaded with the main program.

In the main program of the following example, the EXTERNAL statement indicates that a user-defined function, DABS, is to be used rather than the specific intrinsic function DABS. This user-defined function is subsequently used as an argument in the CALL TRIG statement.

Main Program

```
EXTERNAL DABS
   .
   .
   .
X = DABS (Y)
CALL TRIG (DABS)
```

Subprogram

```
FUNCTION DABS (X, T)
DABS = X - T
RETURN
END
```

The appearance of a generic intrinsic function name in an EXTERNAL statement will override the generic property of that name when it is used in a function reference within the program unit.

```
EXTERNAL ABS
      .
      .
      .
I=ABS (J)
      .
      .
      .
CALL S(ABS,X,Y)
      .
      .
      .
```

In the preceding example, the reference ABS (J) selects the external function ABS rather than the specific intrinsic function IABS. The name ABS is passed as an external procedure to subprogram S.

7.8 INTRINSIC Statement

The INTRINSIC statement identifies a symbolic name as representing an intrinsic function. The name can then be used as an actual argument to a subprogram. The INTRINSIC statement can also verify that symbolic names do, in fact, correspond to intrinsic functions recognized by the compiler.

Syntax

```
INTRINSIC  func [,func]
```

func An intrinsic function name.

Rules for Use

- The name of a specific intrinsic function used as an actual argument in a program unit must appear in an INTRINSIC statement in that program unit.
- The names of intrinsic functions for type conversions (INT, IFIX, IDINT, FLOAT, HFIX, DREAL, SNGL, REAL, DBLE, CMPLX, ICHAR, and CHAR) must not be used as actual arguments.
- The names of intrinsic functions for choosing the largest or smallest value (MAX, MAX0, AMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, AMIN0, DMIN1, DMAX1 and MIN1) must not be used as actual arguments.
- The names of intrinsic functions for logical relationships (LGE, LGT, LLE, LLT) must not be used in actual arguments.
- The appearance of a generic function name in an INTRINSIC statement does not cause that name to lose its generic property.

- A function name should appear only once in all of the INTRINSIC statements of a program unit.
- A function name must not appear in both an EXTERNAL and an INTRINSIC statement within a program unit.

Example

```
INTRINSIC  EXP, MOD, IOR
   .
   .
   .
CALL SUBR (EXP, MOD, IOR)
   .
   .
   .
```

7.9 SAVE Statement

The SAVE statement enables a variable, array, or array element to retain its definition status after the execution of a RETURN or END statement in a subprogram. The variable, array, or array element does not become undefined as a result of the execution of a RETURN or END statement.

Syntax

```
SAVE [a [,a] . . . ]
```

a — A common block name (preceded and followed by a slash), a variable name, or an array name.

Rules for Use

- A SAVE statement is optional in a main program and has no effect.
- A SAVE statement must not contain a dummy argument name, procedure name, or entities within a common block.
- 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 containing a common block name preceded and followed by slashes specifies all of the entities in that common block.
- A SAVE statement specifying a named common block causes the current values of common block entities to be retained at the time a RETURN or END statement is executed. These values are made available to the next program unit that specifies the common block name.

- To comply with the ANSI X3.9-1978 standard, a common block name specified by a SAVE statement in a subprogram must be specified by a SAVE statement in every subprogram in which that common block appears; however, FORTRAN 77+ does not require this.

Examples

```
SAVE /AREA/
SAVE ALPHA
SAVE
```

In the above examples, AREA is a common block and ALPHA is a variable name. The last statement illustrates a SAVE without a list. In this case, the names of all allowable items in the program unit will be saved.

Note: Under FORTRAN 77+, all entities are saved by default.

CHAPTER 8

DATA STATEMENT

8.1 General

The DATA statement initializes variables, arrays, array elements, and substrings.

Syntax

DATA $nlist_1/clist_1/[,nlist_2/clist_2/. . .,nlist_i/clist_i/]$

nlist — A list of variable names, array names, array element names, substring names and implied DO lists.

clist — A list of constants with the following form:

$a_1[,a_2...,a_i]$

where a_i has one of the forms

c

r*c

c — A constant or the symbolic name of a constant (i.e., not a constant expression).

r — A nonzero, unsigned integer constant or the symbolic name of an integer constant that specifies the number of times the same value c is to be assigned to successive entities in the associated nlist.

Constant values in each clist are assigned to the entities in the preceding nlist. Values are assigned in order as they appear, from left to right.

Rules for Use

- The number of constants must correspond exactly to the number of entities in the preceding nlist.
- When an unsubscripted array name appears in a DATA statement, values are assigned to every element of that array. The associated constant list must therefore contain enough values to fill the array. Array elements are filled in the order of subscript progression.
- If both the constant value in clist and the entity in nlist have numeric data types, the conversion is based on the following rules:

  The constant value is converted, if necessary, to the data type of the variable being initialized.

Hexadecimal, octal, and binary constants are stored (unconverted) right-justified with leading zeros within the number of storage units required by the corresponding item. If the number of significant characters in the string is greater than the number of storage units required by the corresponding item, only the rightmost characters will be used to define the constant. For example, an integer halfword can contain only four hexadecimal digits; therefore, in the statement:

```
INTEGER*2 H
DATA H/Z'ABCDEF'/
```

the rightmost characters (CDEF) will be used to define the constant. If more than twice as many characters are specified than required, an error message is issued.

Hollerith constants (of the form nHs) may be used in a DATA statement. The following rules are applicable:

If the number of characters in a Hollerith item is less than the number of storage units required, the constant will be left-justified within the required number of storage units, and the remaining units will be filled with blanks.

For a Hollerith string comprising more than n characters, the leading n characters are stored in the corresponding item. A new storage item is obtained from the data list until the Hollerith constant is exhausted.

An example of initialization of several items with a single Hollerith constant follows:

```
INTEGER M(2)
DATA M/8HABCDEFG¥ø/
```

is the same as

```
INTEGER M(2)
DATA M/4HABCD, 4HEFG¥ø/
```

If a constant is entered that is too large, an error message is issued, and the compiler accepts as many characters as it can, keeping them left-justified.

Right justified Hollerith constants (nRs form) values are stored unconverted and right justified with leading zeros. Any excess characters are truncated from the left.

- If the constant value in clist and the entity in nlist are both of CHARACTER data type, the processing is based on the following rules:

  If the length of the character entity in nlist is greater than the length of its corresponding character constant in clist, the additional rightmost characters of the entity are blank-filled. For example,

  ```
  CHARACTER*4 TOT
  DATA TOT /'ABC'/
  ```

  results in TOT having the value 'ABC␣'

  If the length of the character entity in nlist is less than the length of its corresponding character constant in clist, the excess rightmost characters are ignored. For example,

  ```
  CHARACTER*4 TOT
  DATA TOT /'ABCDE'/
  ```

  results in TOT having the value 'ABCD'

Example 1

```
      REAL MATRIX
      DIMENSION MATRIX (2,2)
      DATA K, MATRIX (1,1), MATRIX (2,1), MATRIX (1,2), MATRIX (2,2) /12,
     X  4.5, 2*6, Z'1FA0'/
```

The DATA statement in the preceding example causes the following:

K will have the value 12
MATRIX (1,1) will have the value 4.5
MATRIX (2,1) will have the value 6.0
MATRIX (1,2) will have the value 6.0
MATRIX (2,2) will have the value X'00001FA0' (base 16)

Example 2

```
DIMENSION NTABLE (3,2)
INTEGER*1 NTABLE
DATA NTABLE, BASE /1,7,4HDATA,2RNE/
```

The DATA statement in the preceding example causes the following:

NTABLE (1,1) will have the value 1
NTABLE (2,1) will have the value 7
NTABLE (3,1) will have the value 44 (base 16)
NTABLE (1,2) will have the value 41 (base 16)
NTABLE (2,2) will have the value 54 (base 16)
NTABLE (3,2) will have the value 41 (base 16)
BASE will have the value X'00004E45' (base 16)

In the preceding example, the appearance of NTABLE within the data list causes the entire array NTABLE to be initialized.

Notice in the following example that a comma in the last line begins a new data statement nlist. Note also that each element of an array may be referenced (as in the case of array H), or an array may be referenced as an entity, for example, array PI.

```
      REAL RTT(2)
      COMPLEX Y
      LOGICAL LT, LF
      DIMENSION H(2,2), PI (3)
      DATA A, B, Y, KO, LT, LF, H(1,1),
     X  H(2,1), H(1,2), H(2,2), PI /5.97,
     X  2.5E-14, (2.73,6.75), 750, .FALSE., .TRUE.,
     X  1.73E-3, 0.8E-1, 2*75.0, 1., 2., 3.141/
     X ,RTT(1) /4HNG-S/
```

8.2 Implied DO in a DATA Statement

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

(dlist, i=m_1, m_2 [,m_3])

dlist	A list of array element names and implied DO lists.
i	The name of an integer variable, called the implied DO variable.
m_1, m_2, m_3	The initial, terminal, and incremental parameters, respectively. These may be integer constant expressions or implied DO variables of other implied DO lists that have this implied DO list within their ranges.

The following rules govern the use of implied DO statements in data statements:

- An iteration count and the values of an implied DO variable are established from m_1, m_2, and m_3 exactly as for a DO loop. However, the iteration count must be greater than zero.
- The statement does not affect the definition status of a variable of the same name as the implied DO variable within the same program unit.

Examples

```
DIMENSION C(3)
DATA A, B, (C (I), I=1,3) /7.5, 6.0, 3*7.0/
```

The preceding DATA statement assigns the following values:

```
A       =   7.5
B       =   6.0
C (1)   =   7.0
C (2)   =   7.0
C (3)   =   7.0
```

The following statement has the same result:

```
DATA  A /7.5/,  B /6.0/,  C /3*7.0/
```

The implied DO in the following DATA statement contains an incremental parameter (m_3) of 2:

```
DIMENSION C(7)
DATA  A, B,  (C(I), I=1,7,2)  / 7.5, 6.0, 4*7.0/
```

Therefore, the following values are assigned:

```
A       =   7.5
B       =   6.0
C (1)   =   7.0
C (3)   =   7.0
C (5)   =   7.0
C (7)   =   7.0
```

```
DIMENSION  C(3,2)
DATA  ( (C (I,J), J=1,2 ), I=1,3)  / 6, 5, 4, 3, 2, 1/
```

The implied DO in the above DATA statement assigns the following values:

```
C (1,1)   =   6
C (1,2)   =   5
C (2,1)   =   4
C (2,2)   =   3
C (3,1)   =   2
C (3,2)   =   1
```

Note that the innermost implied DO is executed first, i.e., J varies from 1 to 2, producing row-major order. The following DATA statement, on the other hand, produces column-major order:

```
DATA  C /6,5,4,3,2,1/
```

The values assigned are:

```
C (1,1)   =   6
C (2,1)   =   5
C (3,1)   =   4
C (1,2)   =   3
C (2,2)   =   2
C (3,2)   =   1
```

The following implied DO contains an example of the implied DO index in an outer loop (M) being used as the terminal parameter in an inner loop:

```
DATA ((Y (M,N), N=1,M), M=1,3)  /6*0./
```

This example has the same effect as executing the following:

```
      DO 1 M=1,3
      DO 1 N=1,M
      Y(M,N)=Ø.
1     CONTINUE
       .
       .
       .
```

The values assigned are:

```
Y (1,1)    =    0
Y (2,1)    =    0
Y (2,2)    =    0
Y (3,1)    =    0
Y (3,2)    =    0
Y (3,3)    =    0
```

CHAPTER 9

FUNCTIONS AND SUBROUTINES

9.1 General

Functions and subroutines are procedures that can be used repeatedly by a program. A procedure may be written once and referenced each time it is required. This chapter discusses the following functions and subroutines:

- Statement functions
- Intrinsic functions
- Function subprograms
- Subroutine subprograms
- Internal functions
- Internal subprograms

A function name reference is considered to be an external reference:

- If the name appears in an EXTERNAL statement

  or

- If it does not appear in an INTRINSIC statement, statement function statement, or INTERNAL FUNCTION statement; and it is not one of the intrinsic function names listed in Tables 9-1 through 9-4.

A subroutine name reference is considered to be an external reference if the name appears in an EXTERNAL statement or if it does not appear in an INTRINSIC statement or INTERNAL SUBROUTINE statement.

9.1.1 Dummy Arguments

Functions and subroutines can be referenced at more than one point throughout a program. Arguments used by a function or subroutine can be different each time the procedure is used. Dummy arguments in functions and subroutines represent the actual arguments that are passed to the procedure when it is called.

Functions and subroutines use dummy arguments to indicate the type of the actual arguments they represent and whether those actual arguments are variables, array elements, arrays, subroutine names or external function names. Each dummy argument must be used within a function or subroutine as if it were a variable, array, array element, subroutine, or external function identifier. Note, however, that a statement function dummy argument can be only a variable. (Examples of argument lists are presented in the following sections.)

Dummy Arguments

When a procedure is referenced, the actual arguments supplied are used where the corresponding dummy arguments appear. Except for subroutine identifiers and literal constants, a valid association between dummy and actual arguments occurs only if both are of the same type; otherwise, the results of procedure executions will be unpredictable. Argument association can be carried through more than one level of procedure reference if a valid association is maintained through each level.

The following rules govern the use of dummy arguments:

- Dummy argument names cannot appear in EQUIVALENCE, DATA, PARAMETER, SAVE, INTRINSIC, or COMMON statements except as common block names.
- A variable dummy argument must have a variable, an array element, a substring, an expression, or a constant as its corresponding actual argument. Note that a constant cannot be used as an actual argument if the variable dummy argument is going to be changed. This would result in the constant being changed.
- An array dummy argument must have either an array name or an array element identifier as its corresponding actual argument. If the actual argument is an array, the length of the dummy array must be less than or equal to that of the actual array. Each element of a dummy array is associated directly with the corresponding elements of the actual array unless the association is started by other than the first array element.
- A dummy argument representing a subroutine identifier must have a subroutine name as its actual argument.
- A dummy argument representing an external function must have an external function as its actual argument.
- A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, ENTRY, or statement function statement in the same program unit.
- If a dummy argument is defined or redefined in a referenced procedure, its corresponding actual argument must be a variable, an array element, or a substring.
- Arguments are passed by reference; i.e., the addresses of the arguments are passed rather than their values.
- If a subprogram has an argument list of more than 250 dummy arguments, set option 7 before compiling the subprogram.

Additional information regarding the use of dummy arguments and actual arguments is given in the descriptions of specific procedures on the following pages.

9.2 Statement Functions

A statement function is a procedure specified in the form of a single arithmetic, logical, or character assignment statement. A statement function can be referenced only within the program unit in which it appears.

Syntax

name ([d_1 [,d_2 . . .[,d_i]]]) = e

name	The symbolic name of the statement function.
d_i	A statement function's dummy argument(s).
e	An arithmetic, logical, or character expression.

Rules for Use

- A statement function statement can only appear after specification statements and before the first executable statement of the program unit in which it is referenced.
- The type of the expression e can be different from the type of the statement function name; however, the relationship between name and e must conform to the assignment rules for arithmetic, logical, and character assignment statements.
- Each primary of the expression e must be one of the following:

  A constant or the symbolic name of a constant

  A variable reference

  An array element reference

  A character substring reference

  An intrinsic function reference

  A reference to a statement function for which the statement function statement appears in the preceding lines of the program unit

  An external function reference

  An internal function reference

  A dummy procedure reference

  An expression enclosed in parentheses that meets all of the requirements specified for the expression e

  A dummy argument reference

- Any statement function name or dummy argument can be explicitly typed in a type specification statement, or it can be implicitly typed.
- The statement function dummy argument list serves only to indicate the order, number, and type of arguments for the statement function.
- Each d is a distinct variable name; however, since each is a dummy argument, it can have the same name as a variable of the same type appearing elsewhere in the program unit.

- The symbolic name of a statement function dummy argument can identify other dummy arguments of the same type in different statement function statements.
- Mismatching of argument lists is permitted.
- The data type of any dummy argument cannot be logical bit.

9.2.1 Referencing a Statement Function

A statement function is referenced by using its name, followed by a parenthesized list of arguments, in an expression. Execution of a statement function reference results in the following:

- Evaluation of actual arguments that are expressions.
- Association of actual arguments with the corresponding dummy arguments.
- Evaluation of the expression e.
- Conversion, if necessary, of an arithmetic expression value to the type of the statement function according to the assignment rules in Chapter 5.

The resulting value is available to the expression that contains the function reference.

When a statement function reference is executed, its actual arguments must be defined. An actual argument list in a statement function reference can be any expression of the same data type as the corresponding dummy argument (except a character expression involving concatenation of a variable or array element whose length specification is an asterisk).

Examples

```
FUNC1 (A,B,C,D) = ((A+B)**C)/D
   .
   .
   .
ZOT = A1 - FUNC1 (X,Y,ZIP,TAN)
```

The following definition is invalid; a statement function dummy argument cannot be a constant:

```
TOTAL (X,Y,Z,40.0) = (A+B+C)/E
```

The following reference is invalid; the data type of the third argument L does not agree in type with the corresponding dummy argument C.

```
FUNC1 (A,B,C,D) = ((A+B)**C)/D
   .
   .
   .
ZOT = A1 - FUNC1 (X,Y,L,Z)
```

9.3 Intrinsic Functions

Intrinsic functions are functions that are built into the FORTRAN 77+ language. Their names are predefined to the compiler and have predefined data types. An intrinsic function is called by referencing its name in an expression.

These functions are listed alphabetically in Tables 9-1 through 9-4. Table 9-1 contains Arithmetic and Conversion Functions, Table 9-2 contains Lexical Comparison Functions, Table 9-3 contains Word and Bit Functions, and Table 9-4 contains Trigonometric Functions.

Syntax

name (a_1 [,a_2 ...[,a_i]])

name — The name of the intrinsic function.

a_i — Actual arguments.

Rules for Use

- All angles are expressed in radians.
- The result of a function of type complex is the principal value.
- The arguments must agree in type, number, and order with the specifications indicated in Table 9-2.
- User-defined external functions having the same names as the functions listed in Table 9-1 must appear in an EXTERNAL statement in the program units in which the functions are referenced.
- The following intrinsic functions will not accept integer *8 arguments:

NOT	FLOAT	IBITS	IOR
BTEST	IAND	IBSET	ISHFT
CHAR	IBCLR	IEOR	ISHFTC

Example:

```
BGST = AMAX1(X,Y,Z,A1)
   .
   .
   .
MAGN1 = ABS (A1)
   .
   .
   .
S3 = SIN (F2)
   .
   .
   .
ROOT = (-B + SQRT (B**2 - 4. *A*C )) / (2.*A)
```

9.3.1 Specific Names and Generic Names

Generic names simplify the referencing of intrinsic functions because the same function name can be used with more than one type of argument. Only a specific intrinsic function name can be passed as an actual argument when the dummy argument is a dummy procedure.

If a generic name is used to reference an intrinsic function, the type of the result (except for intrinsic functions performing type conversion, nearest integer, and absolute value with a complex argument) is the same as the type of the argument.

For intrinsic functions that have more than one argument, all arguments must be of the same type.

If the specific name or generic name of an intrinsic function appears in the dummy argument list of a function or subroutine in a subprogram, that symbolic name does not identify an intrinsic function in the program unit. The data type identified with the symbolic name is specified in the same manner as for variables and arrays.

9.3.2 Inline Intrinsic Functions

The FORTRAN 77+ compiler generates inline code for the intrinsic functions listed below.

ABS	IABS	JABS	DABS	SIGN	ISIGN	JSIGN	DSIGN
DIM	IDIM	JDIM	REAL	DREAL	AIMAG	DIMAG	CONJG
CMPLX	DCMPLX	SNGL	DBLE				

The compiler will not generate inline code under the following circumstances:

- the function is declared EXTERNAL
- the function is declared INTRINSIC and referenced in a subprogram where the name of the function was passed as an argument to the subprogram

In those cases, the FORTRAN 77+ compiler generates a branch and link instruction for those references to the function. Note that the use of inline code results in faster execution at run-time.

Table 9-1 (Page 1 of 7)
Arithmetic and Conversion Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
ABS	IABS	1	integer	integer
	JABS		integer*8	integer*8
	ABS		real	real
	DABS		double precision	double precision
	CABS		complex*8	real
	CDABS		complex*16	double precision
	Returns absolute value (|a|) of any numeric argument.			
			J=IABS(a)	
---	AIMAG	1	complex*8	real
	DIMAG		complex*16	double precision
	Returns imaginary part of complex argument. A complex value is expressed as an ordered pair of reals, (ar,ai), where ar is the real part and ai is the imaginary part.			
			R=AIMAG(a)	
AINT	AINT	1	real	real
	DINT		double precision	double precision
	Returns the truncated value of a real or a double precision argument. If |a| < 1, AINT(a)=0.0; if |a| $\geq$ 1, AINT(a) is the real value whose nonfractional part is equal in magnitude to the largest integer that does not exceed the magnitude of a and whose sign is the same as the sign of a. The fractional part is zero. For example, AINT(-3.7)=-3.0.			
			R=AINT(a)	
ANINT	ANINT	1	real	real
	DNINT		double precision	double precision
	Returns nearest whole number of real or double precision argument. If a $\geq$ 0, ANINT(a)=INT(a+.5); if a < 0, ANINT(a)=INT(a-.5).			
			R=ANINT(a)	

Table 9-1 (Page 2 of 7)
Arithmetic and Conversion Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
---	CHAR	1	integer	character
	Returns the character in the ath position in the ASCII collating sequence. a must be of type integer. The value returned is of type character and has a length of one. ICHAR(CHAR(i))=i for $0 \leq i \leq 255$; CHAR(ICHAR(c))=c for any character c.			
			CH=CHAR(a)	
CMPLX	---	1 or 2	integer	complex
	---	1 or 2	real	complex
	---	1 or 2	double precision	complex
	---	1 only	complex*8 or *16	complex
	Converts one or two numeric arguments to return complex value(s). If there are two arguments, they must be of the same type. For a of type complex, CMPLX(a) is a. For a of type integer, real, or double precision, CMPLX(a) is the complex value whose real part is REAL(a) and whose imaginary part is zero. CMPLX(a_1,a_2) is the complex value whose real part is REAL(a_1) and whose imaginary part is REAL(a_2).			
			Z=CMPLX(a_1,a_2)	
---	CONJG	1	complex*8	complex*8
	DCONJG		complex*16	complex*16
	Returns complex conjugate. For a=X+iY, result is X-iY. A complex value is expressed as an ordered pair of reals, (ar,ai), where ar is the real part and ai is the imaginary part.			
			Z=CONJG(a)	
DBLE	---	1	integer	double precision
	---		real	double precision
	---		double precision	double precision
	---		complex*8 or *16	double precision
	Converts any numeric argument to a double precision value. For a of type double precision, DBLE(a) is a. For a of type integer or real, DBLE(a) is as much precision of the significant part of a as a double precision datum can contain. For a of type complex, DBLE(a) is as much precision of the significant part of the real part of a as a double precision datum can contain.			
			DP=DBLE(a)	

Table 9-1 (Page 3 of 7)
Arithmetic and Conversion Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
DCMPLX	---	1 or 2	integer	complex*16
	---	1 or 2	real	complex*16
	---	1 or 2	double precision	complex*16
	---	1 only	complex*8 or *16	complex*16
	Converts one or two numeric arguments to return complex value(s). If there are two arguments, they must be of the same type. For a of type complex, DCMPLX(a) is a. For a of type integer, real, or double precision, DCMPLX(a) is the complex value whose real part is DBLE(a) and whose imaginary part is zero. DCMPLX(a_1,a_2) is the complex value whose real part is DBLE(a_1) and whose imaginary part is DBLE(a_2).			
			Z=DCMPLX(a_1,a_2)	
DIM	IDIM	2	integer	integer
	JDIM		integer*8	integer*8
	DIM		real	real
	DDIM		double precision	double precision
	Returns positive difference (a_1-MIN(a_1, a_2)) between two non-complex numeric arguments.			
			J=IDIM(a_1,a_2)	
---	DPROD	2	real	double precision
	Returns double precision product (a_1*a_2) of two real arguments.			
			DP=DPROD(a_1,a_2)	
---	EXP	1	real	real
	DEXP		double precision	double precision
	CEXP		complex*8	complex*8
	CDEXP		complex*16	complex*16
	Returns value obtained after exponentiation (e^a).			
			R=EXP(a)	

Table 9-1 (Page 4 of 7)
Arithmetic and Conversion Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
---	ICHAR	1	character	integer
	Converts character argument to return integer value. The position of that character in the ASCII collating sequence is the value of ICHAR. The first character in the collating sequence corresponds to position 0 and the last to position 255; there are 256 characters in the sequence. a must be a character from the ASCII character set and it must have a length of one. For any characters c_1 and c_2 from the ASCII character set, (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.			
			J=ICHAR(a)	
---	INDEX	2	character	integer
	Returns an integer value indicating the starting position within character string a_1 of a substring identical to string a_2. If a_2 occurs more than once in a_1, the starting position of the first occurrence is returned. If a_2 does not occur in a_1, or if string a_2 is longer than string a_1, the value zero is returned.			
			J=INDEX(a_1,a_2)	
INT	---	1	integer	integer
	JINT		integer*8	integer*8
	INT or IFIX		real	integer
	HFIX		real	integer*2
	IDINT		double precision	integer
	---		complex*8 or *16	integer
	Converts any numeric argument to return integer value. For a of type integer, INT(a)=a. For a of type real or double precision: If $\|a\| < 1$, INT(a)=0; if $\|a\| \geq 1$, INT(a) is the integer whose magnitude is the largest integer that does not exceed the magnitude of a and whose sign is the same as the sign of a. For example, INT(-3.7)=-3. For a of type complex, INT(a) is the value obtained by applying the aforementioned rule to the real part of a.			
			J=INT(a)	

Table 9-1 (Page 5 of 7)
Arithmetic and Conversion Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
LEN	LEN	1	character	integer
	Returns the length of character string. The LEN function can determine the length of the variable or constant specified as the argument without that variable or constant having been initialized.			
			J=LEN(a)	
LOCF	LOCF	1	any	integer
	ADDR		any	integer
	Returns the address of the argument. LOCF and ADDR differ in that ADDR returns a 24-bit pure address with the format bits cleared. The ADDR function is generated inline by the compiler and is not available on the SRTL.			
			J=LOCF(a)	
LOG	ALOG	1	real	real
	DLOG		double precision	double precision
	CLOG		complex*8	complex*8
	CDLOG		complex*16	complex*16
	Returns the natural logarithm ($\log_e(a)$) of the argument. The arguments of ALOG and DLOG must be greater than zero. The argument of CLOG or CDLOG must not be (0.,0.). The range of the imaginary part of the result of CLOG or CDLOG is: $-\pi <$ imaginary part $\leq \pi$. The imaginary part of the result is π only when the real part of the argument is less than zero and the imaginary part of the argument is zero.			
			R=ALOG(a)	
LOG10	ALOG10	1	real	real
	DLOG10		double precision	double precision
	Returns the common logarithm ($\log_{10}(a)$) of the argument. The arguments for LOG10 must be greater than zero.			
			R=LOG10(a)	

Table 9-1 (Page 6 of 7)
Arithmetic and Conversion Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
MAX	MAX0	2 or more	integer	integer
	AMAX0		integer	real
	MAX1		real	integer
	AMAX1		real	real
	DMAX1		double precision	double precision
	Returns maximum value of arguments.			
			J=MAX0(a_1, a_2. . .,a_n)	
MIN	MIN0	2 or more	integer	integer
	AMIN0		integer	real
	MIN1		real	integer
	AMIN1		real	real
	DMIN1		double precision	double precision
	Returns minimum value of arguments.			
			J=MIN0(a_1, a_2. . .,a_n)	
MOD	MOD	2	integer	integer
	AMOD		real	real
	DMOD		double precision	double precision
	Returns remainder when a_1 is divided by a_2 with the sign of a_1. The result is undefined when the value of the second argument is zero.			
			J=MOD(a_1,a_2)	
NINT	NINT	1	real	integer
	IDNINT		double	integer
	Returns nearest integer for argument. If a $\geq$ 0, INT(a+.5); if a < 0, INT(a-.5).			
			J=NINT(a)	

Table 9-1 (Page 7 of 7)
Arithmetic and Conversion Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
REAL	REAL	1	integer	real
	FLOAT		integer	real
	---		real	real
	SNGL		double precision	real
	---		complex*8	real
	DREAL		complex*16	real
	Converts any numeric argument to return real value. For a of type real, REAL(a) is a. For a of type integer or double precision, REAL(a) is as much precision of the significant part of a as a real datum can contain. For a of type complex, REAL(a) is the real part of a.			
	R=REAL(a)			
SIGN	ISIGN	2	integer	integer
	JSIGN		integer*8	integer*8
	SIGN		real	real
	DSIGN		double	double
	Returns the magnitude of the first argument with the sign of the second argument. If the first argument is zero, then zero is returned. If the second argument is zero, then a positive value is always returned.			
	J=ISIGN(a_1, a_2)			
SQRT	SQRT	1	real	real
	DSQRT		double precision	double precision
	CSQRT		complex*8	complex*8
	CDSQRT		complex*16	complex*16
	The argument of SQRT and DSQRT must be greater than or equal to zero. The result of CSQRT or CDSQRT is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.			
	R=SQRT(a)			

Table 9-2
Lexical Comparison Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
---	LGE*	2	character	logical
	Returns the value .TRUE. if $a_1=a_2$ or if a_1 follows a_2 in the ASCII collating sequence. Otherwise, returns the value .FALSE.			
			$L=LGE(a_1,a_2)$	
---	LGT*	2	character	logical
	Returns the value .TRUE. if a_1 follows a_2 in the ASCII collating sequence. Otherwise, returns the value .FALSE.			
			$L=LGT(a_1,a_2)$	
---	LLE*	2	character	logical
	Returns the value .TRUE. if $a_1=a_2$ or if a_1 precedes a_2 in the ASCII collating sequence. Otherwise, returns the value .FALSE.			
			$L=LLE(a_1,a_2)$	
---	LLT*	2	character	logical
	Returns the value .TRUE. if a_1 precedes a_2 in the ASCII collating sequence. Otherwise, returns the value .FALSE.			
			$L=LLT(a_1,a_2)$	

* If the argument values for these functions are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

Table 9-3 (Page 1 of 2)
Word and Bit Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
---	BTEST	2	integer	logical
	The bits are numbered 0 through 31, from right to left. BTEST tests the a_2 bit of a_1. If a_2 is set, BTEST=.TRUE. If a_2 is clear, BTEST=.FALSE.			
			L=BTEST(a_1,a_2)	
---	IAND AND	2	integer real	integer real
	Returns result of logical AND (a_1 ^ a_2).			
			J=AND(a_1,a_2)	
--- ---	IBSET IBCLR	2 2	integer integer	integer integer
	The bits are numbered 0 through 31, from right to left. The argument a_1 is not altered. IBSET returns the value of a_1 with the a_2 bit set. IBCLR returns the value of a_1 with the a_2 bit cleared.			
			J=IBSET(a_1,a_2)	
---	IBITS	3	integer	integer
	The bits are numbered 0 through 31, from right to left. The argument a_1 is not altered. The value of a_2+a_3 must be less than or equal to 32. IBITS extracts a bit field from a_1, starting at bit a_2 and extending left for a_3 bits. The result field is right justified and the remaining bits are set to zero.			
			J=IBITS(a_1,a_2,a_3)	
---	IEOR	2	integer	integer
	Returns result of exclusive OR (a_1 ^ a_2).			
			J=IEOR(a_1,a_2)	

Table 9-3 (Page 2 of 2)
Word and Bit Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
---	IOR	2	integer	integer
	Returns result of inclusive OR (a_1 ^{v} a_2).			
			J=IOR(a_1,a_2)	
---	ISHFT SHIFT	2	integer integer	integer integer
	Returns a_1 that has been shifted by a_2 places. $a_2 < 0$ for right shift, $a_2 > 0$ for left shift.			
			J=ISHFT(a_1,a_2)	
---	ISHFTC	3	integer	integer
	The rightmost a_3 bits of a_1 are circularly shifted by a_2 places. The bits shifted out of one end of the subfield a_3 are shifted into the opposite end of the subfield a_3. The shifted bits are combined with the leftmost (32-a_3) unshifted bits to form the function result.			
			J=ISHFTC(a_1,a_2,a_3)	
---	NOT	1	integer	integer
	Returns the bit complement ($\overline{a}$) of the argument in decimal form. For each bit in a, the complement (opposite) binary value is inserted.			
			J=NOT(a)	

Table 9-4 (Page 1 of 2)
Trigonometric Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
ACOS	ACOS	1	real	real
	DACOS		double precision	double precision
	Returns the arc cosine ($\cos^{-1}(a)$) of the argument. The absolute value of a must be less than or equal to one. The range of the result is: $0 \leq$ result $\leq \pi$.			
			X=ACOS(a)	
ASIN	ASIN	1	real	real
	DASIN		double precision	double precision
	Returns the arc sine ($\sin^{-1}(a)$) of the argument. The absolute value of a must be less than or equal to one. The range of the result is: $-\pi/2 \leq$ result $\leq \pi/2$.			
			X=ASIN(a)	
ATAN	ATAN	1	real	real
	DATAN		double precision	double precision
ATAN2	ATAN2	2	real	real
	DATAN2		double precision	double precision
	Returns the arc tangent ($\tan^{-1}(a)$ or $\tan^{-1}(a_1/a_2)$) of the argument. The range of the result for ATAN and DATAN is: $-\pi/2 \leq$ result $\leq \pi/2$. If the value of the first argument of ATAN2 or DATAN2 is positive, the result is positive. If the value of the first argument is zero, the result is zero if the second argument is positive or the result is π if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is $\pi/2$. The arguments cannot both have the value zero. The range of the result for ATAN2 and DATAN2 is: $-\pi <$ result $\leq \pi$.			
			X=ATAN(a)	
COS	COS*	1	real	real
	DCOS*		double precision	double precision
	CCOS		complex*8	complex*8
	CDCOS		complex*16	complex*16
	Returns the cosine (cos(a)) of the argument.			
			X=COS(a)	

Table 9-4 (Page 2 of 2)
Trigonometric Intrinsic Functions

Generic Name	Specific Name	No. of Args.	Argument Data Type	Result Data Type
COSH	COSH DCOSH	1	real double precision	real double precision
	Returns the hyperbolic cosine (cosh(a)) of the argument.			
			X=COSH(a)	
SIN	SIN* DSIN* CSIN CDSIN	1	real double precision complex*8 complex*16	real double precision complex*8 complex*16
	Returns the sine (sin(a)) of the argument.			
			X=SIN(a)	
SINH	SINH DSINH	1	real double precision	real double precision
	Returns the hyperbolic sine (sinh(a)) of the argument.			
			X=SINH(a)	
TAN	TAN* DTAN*	1	real double	real double
	Returns the tangent (tan(a)) of the argument.			
			X=TAN(a)	
TANH	TANH DTANH	1	real double precision	real double precision
	Returns the hyperbolic tangent (tanh(a)) of the argument.			
			X=TANH(a)	

* The absolute value of the argument for these instrinsic functions can be greater than 2π.

9.4 Function Subprogram

A function subprogram is a program unit that has a FUNCTION statement as its first statement, followed by other statements, and terminated by an END statement. It is an independently written program unit that is executed whenever its name is referenced in another program unit. A function subprogram returns a single value to the calling program unit by assigning that value to the function name.

Syntax

[type [* s]] FUNCTION name ([d_1 [,d_2. . . [,d_i]]])

type — Integer, real, double precision, complex, logical, bit, character.

s — One of the permissible length specifications for its associated type.

name — The name of the function subprogram.

d_i — Dummy arguments that represent variable names, array names, subroutine names, or function names.

Rules for Use

- The data type of any dummy argument cannot be BIT.
- The FUNCTION statement must be the first noncomment statement in the program unit.
- The function name within the function acts as a variable that contains the value of the function to be returned to the calling program; it must be assigned a value before a RETURN is executed. In the following sequence of statements, the function name ZED1 appears in an assignment statement that defines the value of the function.

```
FUNCTION ZED1 (A, B, C)
  .
  .
  .
ZED1 = 5. *(A-B) + SQRT(C)
  .
  .
  .
RETURN
  .
  .
  .
END
```

- The symbolic name of a dummy argument must not appear in EQUIVALENCE, COMMON, or DATA statements in the function subprogram, except as a common block name. The names in the dummy argument list may not appear in PARAMETER, SAVE, or INTRINSIC statements in the function subprogram.
- If a dummy argument is an array name, an array declarator must appear in the subprogram with dimensioning information consistent with that in the calling program.

- A character dummy argument whose length specification is an asterisk in parentheses must not appear as an operand for concatenation, except in a character assignment statement.
- The function returns a value that is the same data type as that of the FUNCTION or ENTRY statement through which it was entered. If no data type is given, the function returns an integer or real value depending on the first letter of the function or entry name.
- The logical termination of the execution of a function subprogram is a RETURN statement. The RETURN statement returns the computed value and control to the calling program.
- The END statement specifies the last statement of the subprogram. All function subprograms must be terminated by an END statement. If the END statement is executed, it has the effect of a RETURN statement.
- Alternate returns are not allowed within function subprograms.
- The equal sign of a statement function must appear on the initial line of the statement.
- A function subprogram must not reference itself, directly or indirectly.
- If the type of the function is specified in the function statement, the function name must not appear in a type statement.

Example

```
      FUNCTION  SUM (ED, I, J)
      DIMENSION  ED (I, J)
      SUM = 0.0
      DO 10  K = 1,I
      DO 10  M = 1,J
10    SUM = SUM + ED (K,M)
      RETURN
      END
```

In the preceding example, the value of SUM is returned to a calling program.

9.4.1 Referencing a Function Subprogram

Function subprograms are called whenever the function name (or the name of an entry within the function subprogram) is used in an expression. Such references take the form:

name ([a_1. . .[,a_i]])

name — A function name.

a_i — Actual arguments.

The type of the function name in the function reference must be the same as the type of the function name in the referenced function. The arguments a_i must agree in type and order with the dummy arguments in the FUNCTION statement of the called function subprogram, and they must agree in number except as discussed in section 9.10. The arguments can be any of the following:

A variable name
An array element
A substring
An array name
An expression
A subroutine or function name

If an actual argument is a subroutine or function name, that name must have appeared in an EXTERNAL or INTRINSIC statement to distinguish it from an ordinary variable. The corresponding dummy arguments in the called function subprograms must be used only in subprogram references.

If a function subprogram contains an adjustable array declarator, the actual argument list of the subprogram call must contain not only the actual array name, but also the actual dimensions for all of the adjustable dimensions within the array declarator unless those dimensions are in COMMON.

When a function subprogram is called, program control transfers to the first executable statement following the referenced FUNCTION or ENTRY statement.

9.5 Subroutine Subprograms

A subroutine subprogram is a program unit consisting of a SUBROUTINE statement followed by a series of statements terminated by an END statement. A CALL statement transfers control to a subroutine subprogram, and a RETURN statement returns control to the calling program unit.

When control transfers to the subroutine, the values of any actual arguments in the CALL statement are associated with any corresponding dummy arguments in the SUBROUTINE statement. The statements in the subprogram are then executed.

Syntax

SUBROUTINE name [(d_1 [,d_2. . .,[d_i]])]

name — The name of the subroutine subprogram.

d_i — Dummy arguments that represent variable or array names, other subroutine or function names, or alternate returns.

Rules for Use

- The SUBROUTINE statement must be the first noncomment statement of the subprogram.
- The subroutine subprogram name must not appear within the subroutine in any statement other than the initial SUBROUTINE statement.

- If a dummy argument is an array name, an array declarator must appear in the subprogram with dimensioning information consistent with that in the calling program. Variables in the bounds of an array declarator must be dummy arguments in the argument list, or they must be in COMMON or extended memory.
- Dummy arguments representing values to be determined by the subroutine subprogram and returned to the calling program must appear within the subprogram either on the left side of the equal sign in an assignment statement or in the input list of an input statement.
- A character dummy argument whose length specification is an asterisk in parentheses must not be an operand for concatenation, except in a character assignment statement.
- A subroutine subprogram must not reference itself, directly or indirectly.
- The data type of any dummy argument can not be bit.
- The logical termination of the execution of a subroutine subprogram is a RETURN statement. The RETURN statement returns control to the calling program.
- The END statement specifies the last statement of the subprogram. All subroutine subprograms must be terminated by an END statement. If the END statement is executed, it has the effect of a RETURN statement.

9.5.1 Referencing a Subroutine Subprogram

A subroutine subprogram is invoked by a CALL statement. The form of a CALL statement is:

CALL name [([a_1 [,a_2. . .[,a_i]]])]

name — The name of a SUBROUTINE subprogram, internal subroutine, or entry to a SUBROUTINE subprogram.

a_1,a_2,...,a_i — The actual arguments to be used by the subroutine.

If an argument list is specified, the CALL statement associates the values in the list with the dummy arguments in the subroutine. It then transfers control to the first executable statement following the named entry point in the subroutine. Parameters are passed by reference, i.e., their addresses are passed rather than their values.

Arguments in the CALL statement must agree in type and order with the corresponding dummy arguments in the subprogram-defining SUBROUTINE, INTERNAL SUBROUTINE, or ENTRY statement.

The arguments in a CALL statement must also comply with the following rules:

- FUNCTION and SUBROUTINE names in the argument list must have previously appeared in an EXTERNAL or INTRINSIC statement.
- An alternate return parameter for a subroutine subprogram is indicated by an asterisk (*) or a dollar sign ($) followed by a statement label. Alternate returns are not permitted from an internal procedure.

- If the called subroutine contains a variable array declarator, the CALL statement must contain the actual name of the array and the actual dimension specifications as arguments. Actual dimension specifications can also be passed in COMMON elements or in an EXTENDED BLOCK.

- If an item in the dummy argument list is an array, the corresponding item in the CALL statement argument list must be an array.

Examples

```
CALL COUNTP
CALL LISTED (A,N,'ABCD')
CALL FFT (A,B,*10,*20,C)
```

If an actual argument is a subroutine or function name, that name must have appeared in an EXTERNAL or INTRINSIC statement to distinguish it from an ordinary variable. The corresponding dummy arguments in the called function subprograms must be used only in subprogram references.

If a subroutine subprogram contains an adjustable array declarator, the actual argument list of the subprogram call must contain not only the actual array name, but also the actual dimensions for all of the adjustable dimensions within the array declarator unless those dimensions are in COMMON or in an EXTENDED BLOCK.

Arguments must agree in type and order, and they must agree in number except as noted in section 9.10.

If an argument list is specified, the CALL statement associates actual arguments with the corresponding dummy arguments in the SUBROUTINE or ENTRY statement.

When a subroutine subprogram is called, program control transfers to the first executable statement following the referenced SUBROUTINE or ENTRY statement.

Example

Main Program

```
DIMENSION X(200), Y(100)
      .
      .
      .
K = 100
CALL TRANS (X, Y, K)
      .
      .
```

Subroutine Subprogram

```
      SUBROUTINE TRANS (A,B,N)
      DIMENSION A(100), B(100)
      DO 10 I = 1, N
10    B(I) = A (I)
      RETURN
      END
```

In the preceding example, the main program (or calling program) calls the subroutine TRANS. The object of the subprogram is to copy part of one array directly into another.

The CALL statement transfers control to the subroutine subprogram TRANS and associates the dummy variables (A, B, N) with the actual arguments that appear in the CALL statement.

9.6 Internal Procedures (Function and Subroutine)

Internal procedures (function and subroutine) perform the same function as external functions and subroutines; however, unlike external procedures, an internal procedure is a closed routine compiled within its host program unit. They share the data environment of the program unit and can be referenced at any point within the program unit where a subprogram call or a function reference is allowed.

Syntax

INTERNAL FUNCTION name ([d_1 [,d_2...[,d_i]]])

INTERNAL SUBROUTINE name [([d_1 [,d_2...[,d_i]]])]

name — The symbolic name of an internal procedure (function or subroutine).

d_i — Dummy arguments that represent variable names, array names, subroutine names, or function names.

Rules for Use

- An internal procedure can be referenced at any point within the host program unit; however, it can not be referenced from within the internal procedure itself.
- Execution of a GOTO statement to transfer control into or out of the body of an internal procedure is improper.
- Statement numbers within an internal procedure must be unique within the host program unit.
- ENTRY statements, alternate returns, another internal procedure or a statement function, and specification statements other than type or DIMENSION can not be used in an internal procedure.
- Internal procedures must be located after all host specification statements and all statement function statements.
- The data type of any dummy argument can not be bit.
- If a dummy argument is an array name, an array declarator must appear in the subprogram with dimensioning information consistent with that in the calling program.
- A character dummy argument whose length specification is an asterisk in parentheses must not appear as an operand for concatenation, except in a character assignment statement.
- Internal procedures must not be defined within a SELECT CASE statement.
- An internal procedure with the same name as an intrinsic function should be declared before any reference is made to it.

9.6.1 Dummy Arguments

An internal procedure can have dummy arguments; otherwise, any variable referenced within the internal procedure is the same as any variable of the same name in the host program unit. Dummy arguments within an internal procedure can be given attributes by means of type and DIMENSION statements immediately following the INTERNAL FUNCTION or INTERNAL SUBROUTINE statement.

9.6.2 Referencing Internal Procedures

An internal procedure is composed of statements beginning with an INTERNAL SUBROUTINE or INTERNAL FUNCTION statement and ending with an END INTERNAL statement.

Control is transferred into an internal procedure only by a CALL statement (for a subroutine) or by a function reference using the name of the internal procedure (for a function).

Control leaves an internal procedure through a RETURN or an END INTERNAL statement. Execution of a GOTO statement to transfer control into or out of the body of an internal procedure is improper.

Format statements anywhere within a host program unit can be referenced at any point within or outside of an internal procedure.

An internal procedure having the same name as an intrinsic function should be declared before it is referenced.

Example

```
PROGRAM MAIN
REAL X,Y,F
     .
     .
     .
CALL S(K,L)                    !ISUM GETS VALUE K+L
     .
     .
     .
X=F(Y)
     .
     .
     .
INTERNAL SUBROUTINE S(I,J)
 ISUM=I+J
END INTERNAL
INTERNAL FUNCTION F(Z)
REAL Z
IF (Z.GT.0) THEN
   F=Z*Z
ELSE
   F=0
END IF
END INTERNAL
END
```

9.7 ENTRY Statement

The ENTRY statement provides multiple entry points within a subprogram. The normal entry point is to the first executable statement within a subroutine or function subprogram. An alternate method of entering a subprogram is to use a CALL statement (to reference subroutine subprograms) or a function reference (to reference function subprograms) that refers to an entry name in the subprogram. In this case, control is passed to the first executable statement following the associated ENTRY statement. Any number of ENTRY statements can be contained in a subprogram.

Syntax

ENTRY name [([d_1[,d_2. . .[,d_n]]])]

name An entry name.

d_i Dummy arguments.

Rules for Use

- Any ENTRY statement must follow all specification statements.
- Entry names in a FUNCTION subprogram can return function results; within a FUNCTION subprogram, all variables whose names are the same as entry names or the names of the FUNCTION subprograms are automatically equivalenced (i.e., occupy the same storage element).
- An entry name can be used in an EXTERNAL statement in the same manner as a function or subroutine name.
- ENTRY statements must not be used within the range of a DO loop, nor in a block IF, SELECT CASE construct, or internal procedure.
- Within a subprogram, two or more entry points can not have the same name.
- If an adjustable array name is in the entry argument list, all variables used in its dimension declarators must also be within the argument list, or they must be in COMMON or in an EXTENDED BLOCK.
- Entry dummy arguments need not match dummy arguments within other ENTRY, FUNCTION, or SUBROUTINE statements in the subprogram, with respect to order, number, or type.
- Alternate returns from a subroutine subprogram entered by means of an entry name are discussed in 9.8.1.
- Statement functions can follow an ENTRY statement only if the statement function precedes the first executable statement of the subprogram.
- If an entry name in a function subprogram is of type character or bit, each entry name and the name of the function subprogram must be of type character or bit, respectively. If the name of the function subprogram or any entry in the subprogram has a length of (*) declared, all such entities must have a length of (*) declared; otherwise, all such entities must have a length specification of the same integer value.

Examples

Main Program

```
CALL SUBB (X,Y,Z)
   .
   .
   .
```

Subprograms

```
SUBROUTINE SUB (A,B,C)
        .
        .
        .
RETURN
ENTRY SUBB (Q,R,S)
        .
        .
        .
RETURN
```

In the preceding example, the CALL is to an entry point SUBB within the subroutine SUB. Execution begins with the first statement following ENTRY SUBB (Q,R,S), using the actual arguments (X,Y,Z) passed in the CALL statement.

9.7.1 ENTRY Association

All variables whose names are also the names of entries within a function subprogram are equivalenced. These variables are also equivalenced with the variable whose name is also the name of the function subprogram. Therefore, any such variable that becomes defined (i.e., is assigned a value) causes all equivalenced variables of the same type to become defined; all equivalenced variables of different type become undefined.

Such variables are not required to be of the same type unless the type is character or bit; however, the variable whose name is used to reference the function must be in a defined state when a RETURN or END statement is executed in the subprogram.

9.8 Function and Subroutine Subprograms Returns

The logical termination of the execution of a function or subroutine subprogram is a RETURN statement, which transfers control to the calling program. The form of the RETURN statement is:

```
RETURN
```

When a RETURN is executed in a function, control is returned to the calling program or to the statement that contains the function reference. When a RETURN statement is executed in a subroutine, control is returned to the next executable statement in the calling program that could logically follow the CALL statement.

Subprograms can contain more than one RETURN statement; however, a subprogram need not contain a RETURN statement. The END statement acts as a RETURN statement in a subprogram.

Examples

```
      SUBROUTINE ROAM (A,B,C)
      READ (3,7) B
      A = B**C
      RETURN
7     FORMAT (F9.2)
      END
```

In the preceding example, control is returned to the calling program at the first executable statement following the CALL ROAM statement.

In the following example, the subroutine subprogram RETURN statement transfers control to the first executable statement in the main program following the call, i.e., in this case, the WRITE statement.

Main Program

```
      DIMENSION X(100), T(50)
            .
            .
            .
            .
      CALL NEW (X,50,BEG)
      WRITE (6,20) X (1),X(50),BEG
20      FORMAT (1X, 3F12.3)
```

Subroutine Subprogram

```
      SUBROUTINE NEW (LIST, N, TOT)
      DIMENSION LIST (100)
            .
            .
            .
            .
      RETURN
      END
```

During execution of a program, a function or subroutine subprogram must not be referenced a second time without the prior execution of a RETURN or END statement in that procedure.

9.8.1 Alternate Returns

Alternate returns enable the user to transfer control to a statement in the calling program other than the one immediately following the subroutine call. The form of an alternate return is:

```
RETURN e
```

where e is an integer expression.

The value of e indicates that the eth alternate return (as specified from left to right) in the actual argument list is to be taken. Alternate returns in the actual argument list are denoted by an asterisk (*) or a dollar sign ($) immediately preceding the statement label in the argument list (refer to the examples below). Note that the subroutine program unit must contain a corresponding asterisk (*).

Alternate returns return control to the statement in the calling program associated with the statement label specified in the main program. The position of the statement label in the main program corresponds to the value of the integer expression e in the RETURN statement.

Example

Main Program

```
      CALL FIRST (A,B,*20,*30,C)
        .
        .
        .
20    A=B
        .
        .
30    Y=SIN(X)
        .
        .
        .
```

Subroutine Subprogram

```
      SUBROUTINE FIRST (X,Y,*,*,Q)
        .
        .
        .
      IF(Z) 60, 70, 80
60    RETURN
70    RETURN 1
80    RETURN 2
      END
```

In the preceding example, the SUBROUTINE statement argument list contains two dummy alternate return arguments, corresponding to the actual arguments *20 and *30 in the CALL statement argument list. The RETURN taken depends on the value of Z, as computed in the subroutine. Thus, if Z is less than 0, the normal return is taken; if equal to 0, return is to statement label 20 in the main program; if greater than 0, return is to statement label 30 in the main program.

Generally, if n is the number of asterisks in the subprogram or ENTRY statement that specifies the currently referenced name, and e is the integer expression in the RETURN statement then:

- If e is not specified in a RETURN statement, or if the value of e is less than one or greater than n, control returns to the statement following the CALL statement that initiated the subprogram reference, thus completing the execution of the CALL statement.

- If $1 \leq e \leq n$, the value of e identifies the eth asterisk in the dummy argument list. Control is returned to the statement identified by the alternate return specifier in the CALL statement that is associated with the eth asterisk in the dummy argument list of the currently referenced name, thus completing the execution of the CALL statement.

Alternate return locations can be specified in an ENTRY statement. For example,

```
SUBROUTINE SUB (T,*,*)
        .
        .
        .
ENTRY SUBA (J,K,*,*,Y)
        .
        .
        .
RETURN 1
        .
        .
        .
RETURN 2
END
```

A CALL issued to entry point SUBA must include actual alternate return arguments. For example,

```
CALL SUBA (N,M,*50,*150,G)
```

In this case, RETURN 1 transfers control to the statement labelled 50, and RETURN 2 transfers control to the statement labelled 150 in the calling program.

9.9 Processing Arrays in Subprograms

If a calling program passes an array name to a subprogram, the subprogram must contain the dimension information pertinent to the array, or the calling program must provide this information. A subprogram must contain array declarators for any of its dummy arguments that represent arrays.

For example, a function subprogram designed to compute the average of any one-dimensional array might be the following:

Calling Program

```
      .
      .
      .
DIMENSION ZAP(50), ZOT(25)
A1 = AVG (ZAP,50)
A2 = A1-AVG (ZOT,25)
```

Function Subprogram

```
      FUNCTION AVG (ARRAY,I)
      DIMENSION ARRAY(I)
      SUM=0.0
      DO 20 J=1,I
20    SUM = SUM+ARRAY(J)
      AVG = SUM/I
      RETURN
      END
```

Note: Actual arrays to be processed by the function subprogram are dimensioned in the calling program. The array names and their actual dimensions are transmitted to the function subprogram by the function subprogram reference. The function subprogram contains dummy array and subscript names and an adjustable array declaration.

Dimension information can be provided to a subprogram in common, as illustrated in the following example:

```
DIMENSION ZAP(50), ZOT(25)
INTEGER ARRAYSIZE
COMMON  /ARRAYDATA/ARRAYSIZE

  .
  .
  .
ARRAYSIZE = 50
A1 = AVG(ZAP)
ARRAYSIZE = 25
A2 = AVG(ZOT)
  .
  .
  .
```

```
FUNCTION AVG(ARRAY)
INTEGER ARRAYSIZE
COMMON
/ARRAYDATA/ARRAYSIZE
DIMENSION ARRAY(ARRAYSIZE)
  .
  .
  .
SUM = 0.0
DO 20 J=1,ARRAYSIZE
SUM = SUM + ARRAY(J)
20 CONTINUE
AVG = SUM/ARRAYSIZE
RETURN
END
```

9.10 Processing of Arguments for Subprogram Calls

The FORTRAN 77+ compiler handles subprogram calls by passing the addresses of the arguments from the calling routine to the corresponding dummy arguments of the subprogram. This is accomplished by one of two methods: inline argument processing or calls to F.PR, a Scientific Run-Time Library routine.

The default method is inline argument processing. However, if any of the dummy arguments are declared as EXTENDED DUMMY, the arguments will be processed by using a call to F.PR. Also, setting option 7 before compiling the subprogram causes the system to process the arguments using a call to F.PR.

Inline argument processing is the most efficient method of transferring the arguments between the calling routine and the subprogram. While this technique is very fast, the program size can increase if the subprogram has a large dummy argument list.

Therefore, inline argument processing should be used for procedures with fewer than sixteen arguments. When the number of arguments is sixteen or more, the speed increase provided by inline argument processing becomes negligible. In such cases, you should set option 7 so that F.PR is called.

9.11 Mismatched Argument Lists

FORTRAN 77+ usually requires that the argument list of a called subprogram match the argument list of the calling program in type, order, and number of parameters. However, it is possible for the number of parameters present in the calling list to be less than the number of dummy parameters in the called program parameter list, subject to the following rules:

- Option 7 must be set before compilation.
- A one-argument entry point must be referenced with exactly one argument.
- An entry point with more than one argument can not be called with fewer than two arguments.
- For at least the initial call to a subprogram, the number of actual arguments must be equal to the number of dummy arguments in the called subprogram in order to give defining values to all arguments. For subsequent calls, the previously defined association of any unspecified arguments (i.e., trailing arguments, but not arguments missing from the beginning or middle of the list) will be retained for use within the called subprogram.

Example

Main Program

```
A=B=C=2.0
CALL SUB (A,B,C)
  .
  .
  .
C = 1.0
CALL SUB (A,B)
  .
  .
  .
```

Subroutine Program

```
SUBROUTINE SUB (X,Y,Z)
  .
  .
  .
T = X*Y*Z
  .
  .
  .
```

In the previous example, the dummy argument Z will be associated with the variable C when the initial call to SUB is performed. For the second call to SUB, Z will retain its association with the variable C. Since C has been redefined prior to making the call, the value 1.0 will be used for Z in the calculation X*Y*Z.

CHAPTER 10

BLOCK DATA SUBPROGRAMS

10.1 Introduction

A block data subprogram initializes data in one or more common blocks during the loading of a FORTRAN 77+ object program.

A block data subprogram begins with the BLOCK DATA statement, followed only by DATA, COMMON, DIMENSION, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, or type statements associated with the data being defined. Comment lines are permitted. The last statement in a block data subprogram is the END statement.

10.2 BLOCK DATA Statement

The BLOCK DATA statement has the following form:

Syntax

```
BLOCK DATA [name]
```

name — The symbolic name of the block data subprogram in which the BLOCK DATA statement appears.

Rules for Use

- A block data subprogram without a name cannot be referenced, nor can it be selected from an object library. Only one block data subprogram without a name is allowed in an executable program.
- If any element in a common block is to be initialized, all elements of the block must be listed in a COMMON statement, even though they might not all be initialized (refer to the following example).
- Data in more than one common block can be initialized in one block data subprogram (refer to the following example).
- Blank common cannot be initialized; only data within named common blocks can be initialized.
- Global common and DATAPOOL cannot be initialized by a block data subprogram.
- A block data name can be used for external identification, for example, in an EXTERNAL statement within a program unit (e.g., a MAIN program); this will force the inclusion of the named block data subprogram from a subroutine library (refer to the MPX-32 Reference Manual).
- A common block cannot be initialized in two block data subprograms.

Example

```
      BLOCK DATA  PNAME
      LOGICAL A1
      DOUBLE PRECISION C
      COMMON/BETA/ B(3,3)/GAM/C(4)
      COMMON/ALPHA/A1,E,G,D
      DATA ((B(I,J), I=1,3), J=1,3)/
     X  1.1, 2.5, 3.8, 3*PI
     X  2*0.52, 1.1/,C(1), C(2), C(3), C(4)/
     X  1.2D0, 5.6D0, 2*1.D0/
      DATA A1/.TRUE./,E/ -5.6/
      END
```

Elements G and D are included in the COMMON statement even though they are not being initialized; also, the BLOCK DATA subprogram is initializing more than one common block.

CHAPTER 11

INPUT/OUTPUT

11.1 Introduction

FORTRAN 77+ provides a set of statements to control and define transmission of data between a compiled program and data handling devices such as card readers, line printers, magnetic tapes, and direct access mass storage devices. The data that are to be transferred can belong to a file or device.

The transmission can be performed using many methods. Formatted or unformatted are the methods most commonly recognized. Refer to Appendix A for additional information on input/output methods.

The FORTRAN 77+ input/output statements have been organized into three groups for discussion:

- READ, WRITE, PRINT, PUNCH, TYPE, and ACCEPT statements that cause specified data to be transmitted between data handling devices and computer internal storage.
- Auxiliary input/output control statements that:
  1) position magnetic tapes and disc files,
  2) establish correspondence between unit numbers (logical file codes) and specific devices or files, and
  3) determine attributes of connections to files or devices.
- FORMAT statements that specify conversion and editing of data between internal and external (character string) form in conjunction with formatted record transmission.

NOTE: A function reference appearing in any part of an input/output statement must not cause the statement to be executed; i.e., a recursive input/output statement reference.

The first two groups are discussed in this chapter. FORMAT statements are discussed in Chapter 12.

11.2 Records

A record is a sequence of values or a sequence of characters. It does not necessarily correspond to a device's physical record. There are three kinds of records:

- Formatted
- Unformatted
- Endfile

A formatted record consists of a sequence of characters. Its length is measured in characters and can be zero. To insure predictable results, formatted records must be read from or written to by formatted input/output statements. These records can be prepared by means other than FORTRAN 77+, such as keypunch or text editor.

An unformatted record consists of a sequence of values. It can contain character and noncharacter data, or it can contain no data. Its length is measured in bytes and depends on the external medium and the input/output list used when it is written. The length can be zero. To insure predictable results, these records must be read from or written to by unformatted input/output statements.

An endfile record is written by an ENDFILE statement. Execution of an ENDFILE statement causes a file mark to be written on the specified file or device if applicable.

11.3 Files

A file is a sequence of records and can be either an internal or external file.

An internal file provides for transferring and converting data between two internal storage areas. Internal files have the following properties:

- The file is a character variable, character array element, character array, or character substring.
- A record of an internal file is a character variable, character array element, or character substring.
- If the file is a character variable, character array element, or character substring, it consists of a single record whose length is the same as the length of the variable, array element, or substring, respectively. If the file is a character array, it is treated as a sequence of character array elements. Each array element is a record of the file, and the ordering of the records of the file is the same as the ordering of the elements in the array. Every record of the file has the same length, which is the length of an element in the array.
- The variable, array element, or substring that is the record of the internal file is defined by writing the record or by other means; e.g., the record can be defined by a character assignment statement. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks.
- A record can be read only if the variable, array element, or substring that is the record is defined.
- An internal file is always positioned at the beginning of the first record prior to data transfer.

Internal files have the following restrictions:

- Records must be read from and written to by sequential access formatted input/output statements that do not specify NAMELIST.
- An auxiliary input/output statement cannot specify an internal file.

11.4 Sequential and Direct Access Methods

An internal file can only be accessed sequentially, while an external file can be accessed sequentially or directly. Some files may have more than one allowed access method; other files may be restricted to one. The set of allowed access methods depends on the file. The method of accessing a file is determined when the file is connected to a unit.

When connected for sequential access, a file has the following properties:

- The order of the records is the order in which they were written; however, if the file can also be accessed directly, the order of the records is the same as that specified for direct access. A record that has not been written since the file was created must not be read.
- The records of the file are either all formatted or all unformatted. The last record of the file, however, can be an endfile record.
- The records of the file cannot be read or written by direct access input/output statements.

When connected for direct access, a file has the following properties:

- The order of the records is the order of their record numbers. However, the records can be read from or written to in any order, regardless of their record numbers. Any record can be written to a file while it is connected to a unit. For example, it is permissible to write record 3, even though records 1 and 2 have not been written. Any record can be read from the file while it is connected to a unit, provided that the record was written since the file was created.
- The records of the file are either all formatted or all unformatted. If the file can also be accessed sequentially, its endfile record (if any) is not considered to be part of the file while it is connected for direct access. If the file can only be accessed directly, the file cannot contain an endfile record.
- Only direct access input/output statements can be used to read and write records.
- All records of the file have the same length.
- Each record of the file is uniquely identified by a positive integer called the record number.
- The records of the file must not be read or written using list-directed or NAMELIST formatting.

11.5 Blocked and Unblocked Files

External files can be either blocked or unblocked.

Blocking is the ability of the MPX-32 operating system to place multiple logical records into a 768-byte physical record, along with record control information.

With unblocked files, the file record size is the full 768-byte physical record. The default for files under MPX-32 is blocked, but for FORTRAN 77+ the default is unblocked.

When performing an input or output operation on an external file, MPX-32 reads or writes one logical record. If the file has been assigned as unblocked, the logical record will be a full 768-byte physical record.

If I/O is performed using unblocked files, record management must be done by the user. List-directed formatting and NAMELIST do not operate on unblocked files.

11.6 Control Information List

A control information list is a list that must include:

- A unit specifier that references the source or destination of the data to be transferred:

  [UNIT=]u

  and may include:

- A format specifier that indicates editing processes:

  [FMT=]b

- Error and end-of-file specifiers that determine the execution sequence if an error occurs or an end-of-file is encountered:

  ERR = c
  END = d

- A record specifier that identifies a record:

  REC = r

- An input/output status specifier that provides the return of the input/output status:

  IOSTAT = ios

The following general rules govern the use of the control information list:

- A control information list must contain only one unit specifier and at most one format specifier, one record specifier, one input/output status specifier, one error specifier, and at most one end-of-file specifier.

- A control information list cannot contain both a record specifier and an end-of-file specifier.

- If the control information list contains a format specifier, the statement is a formatted or a NAMELIST input/output statement; otherwise, it is an unformatted input/output statement.

- If the control information list contains a record specifier, the statement is a direct access input/output statement; otherwise, it is a sequential access statement.

Additional information concerning control information list items is provided in the discussion of each input/output statement.

11.6.1 The Unit Specifier

A data transfer input/output statement that contains a control information list includes a parameter that indicates an external unit or internal file. An external unit identifier refers to an external unit, while an internal unit identifier refers to an internal file.

The unit specifier is composed of the keyword UNIT= and the constant or variable u, which is called the unit identifier. The keyword is optional; however, the unit identifier is required.

An external unit identifier can be one of the following:

- An integer expression whose value must be either between 0 and 999, or a left-justified, blank-filled string of one to three ASCII characters (referred to as a logical file code) contained in an INTEGER*4 variable.
- A logical file code constant in the form 's', where s is a string of one to three characters.
- An asterisk designating a default assignment for the external unit. An asterisk cannot be used as a value in an auxiliary input/output statement. Possible default assignments are:

Statement	Default Assignment	
READ	'SI'	
ACCEPT	'UT'	(interactive operation only)
WRITE	'LO'	
PRINT	'LO'	
PUNCH	'BO'	
TYPE	'UT'	(user terminal for interactive operation; System Listed Output (SLO) for batch or independent operation)

Note that PRINT, PUNCH, ACCEPT, and TYPE do not specify a unit; it is implied.

An internal file identifier is the name of a character variable, character array, character array element, or character substring.

Physical data handling units (discs, tapes, etc.) must be assigned to their corresponding unit identifiers before execution of the I/O or auxiliary I/O operations. This assignment can be made using job control language or an OPEN statement. Direct access files must be explicitly opened by an OPEN statement before reading or writing. Refer to the MPX-32 Reference Manual for a discussion of $ASSIGN statements and how they are used to establish the correspondence between unit identifiers and devices or files.

11.7 Input/Output List

An input/output list specifies the names of variables, arrays, array elements, and character substrings to which input data are to be assigned or from which data are to be written. For an output data list, list elements can be expressions.

Single Datum/Multiple Data Identifiers

Input/output lists have the form:

$m_1, m_2, \ldots, m_i$

m_i List items separated by commas.

An input/output list can contain variable names, subscripted array names, unsubscripted array names, or array names accompanied by indexing specifications in a form called an implied DO. Constants, function references, and expressions (except BIT type) can be present in an output list. Data names of type BIT cannot appear as list items in READ/WRITE statements. Function references must not cause definition of any other item in the same output list.

11.7.1 Single Datum Identifier

A single datum identifier item is the name of a variable, array element, or character substring.

The following are single datum identifier lists:

```
A
ALPHA, I (10,10) , M (10,9), SAM
JOHN (5:11)
```

11.7.2 Multiple Data Identifiers

There are two forms of multiple data identifer items:

- An array name appearing in a list without subscript(s) is considered equivalent to the ordered listing of each successive array element. For example, if B is an array with two dimensions, the list item B is equivalent to list items:

  B (1,1), B (2,1),. . .,B (J,K)

  where J and K are the subscript limits of B.

- Implied DO items are lists of one or more single datum identifiers or other implied DO items, followed by a comma and an expression of the form:

  $i = m_1, m_2 [,m_3]$

  enclosed in parentheses.

  The elements i, m_1, m_2, and m_3 have the same meaning as defined for the DO statement; however, m_1, m_2, and m_3 cannot be expressions. An implied DO applies to all list items enclosed in parentheses with the implied DO.

Examples of implied DO lists follow:

Implied DO Lists	Equivalent Lists
(X(I), I=1,4)	X(1), X(2), X(3), X(4)
(Q(J), R(J), J=3,4)	Q(3), R(3), Q(4), R(4)
(P(K), K=2,8,3)	P(2), P(5), P(8)
((A(I,J), I=3,4),J=2,8,3)	A(3,2), A(4,2), A(3,5), A(4,5), A(3,8), A(4,8)
ABLE, (R(L), L=1,3),BETA	ABLE,R(1), R(2), R(3), BETA
(A,Y(M), M=1,3)	A,Y(1), A, Y(2), A,Y(3)

All or a portion of an array can be transmitted through the use of implied DO lists.

The following notes further define input/output list specifications:

- The ordering of a list is from left to right.
- Items enclosed in parentheses (other than as subscripts) with controlling implied DO index parameters are repeated until the index parameters are exhausted.
- Entire arrays can be transmitted using the array name (unsubscripted) in an input/output list.
- For input lists, the implied DO index parameters (i, m_1, m_2, and m_3) cannot appear within the parentheses as list items.

  READ (1,20) (I,J,A(I), I=1,J,2) is not valid

  READ (1,20) I,J,(A(I), I=1,J,2) is valid

  WRITE (1,20) (I,J,A(I), I=1,J,2) is valid

- Any number of items can be in a single list.
- In formatted transmission (i.e., READ (u,b) list or WRITE (u,b) list), each list item must be of the correct type, as specified by the FORMAT statement.

11.8 Control and Interpretation of Data

Data transmission is controlled and interpreted in one of the following ways:

- Unformatted
- Formatted
- NAMELIST
- List-directed

In unformatted mode, the data are transferred between program variables and a file without any data conversion; no FORMAT statement is involved. That is, the data in the file are in internal, machine-dependent form.

In formatted mode, the data in the file are typically in character form. READ and WRITE statements in formatted mode perform a specified translation of each data item between its internal and character representations, as directed by specifiers supplied in the FORMAT statement.

In NAMELIST mode, the external representation of each data item includes its program variable name along with the character representation of the data item.

In list-directed mode, the need to be concerned with card columns, line boundaries, and format statements is eliminated. The data to be transferred and the type of each transferred datum are specified by the contents of an input/output list included in the input/output command.

11.9 Input/Output Statements

Input/output statements are used to store and retrieve data in a fixed order. These statements are device independent and can be used for files on any device.

The data transfer input/output statements include: READ, ACCEPT, WRITE, TYPE, PRINT and PUNCH.

11.9.1 Input Statements

READ and ACCEPT statements transfer data from peripheral devices into specified program variables, arrays, or array elements.

Syntax

```
READ ( [UNIT=] u [, [FMT=] b] [,END=c] [,ERR=d] [,REC=r] [,IOSTAT=ios] )  [list]
  or
READ b [,list]
  or
READ nl
  or
READ (u,nl)
  or
ACCEPT b [,list]
  or
ACCEPT nl
```

u — A unit specifier as described in section 11.6.1

b — A format specifier; b is a statement label or array name identifying the FORMAT statement that describes the record(s) being read. This identifier can also be a character array name or character expression that contains the format information. This parameter can also be an asterisk (*) specifying list-directed formatting.

nl — A NAMELIST name as described in section 11.9.3.

c — An end-of-file specifier; c is the statement label to which control is transferred when the end of the input file is encountered.

d — An error specifier; d is the statement label to which control is transferred when an error condition is encountered during data transfer.

r — A record specifier; r is the number of the record to be read. This specifier is used for direct access input/output.

ios — An input/output status specifier; ios is an integer word variable or array element. Execution of an input statement containing this specifier causes ios to become defined with a zero if no error condition exists, with a positive integer if an error condition exists, or with a negative value if an end-of-file condition is encountered and no condition error exists. IOSTAT values are listed with the execution-time diagnostics in Appendix D.

list — An input/output list (refer to 11.7).

Rules for Use

- If the optional characters UNIT= are omitted from the unit specifier, the unit identifier u must be the first item in the control information list.
- If the optional characters FMT= are omitted from the format specifier, the format identifier b must be the second item in the control information list; the unit specifier must be the first item.
- The specifiers END= and REC= cannot appear in the same statement.
- ACCEPT can only be used interactively.
- If an asterisk is used as a format identifier, the format is determined by the data (list-directed). This eliminates the requirement for putting input data in certain columns, since the data can be written in the form of FORTRAN constants, separated by commas, blanks, or slashes (refer to 11.10).
- Parameters that are specified with keywords can be in any order; for example, END=c and ERR=d can be reversed within the parentheses.
- The following notes define END=, ERR=, and IOSTAT= processing.

  I/O Completes Normally:

  Control returns to the next executable statement. If IOSTAT= was specified, IOS = 0.

EOF or EOM Detected:

If neither END= nor IOSTAT= was specified, a run-time message is generated and the task is aborted.

If only IOSTAT= was specified, program control goes to the next executable statement. The user is responsible for testing IOS and taking appropriate action.

If only END= was specified, program control goes to the statement specified in END=.

If END= and IOSTAT= were both specified, program control goes to the statement specified in END=, and status is sent to IOS.

I/O Error Detected (note that the position of the file becomes indeterminate):

If neither ERR= nor IOSTAT= was specified, a run-time message is generated and the task is aborted.

If only IOSTAT= was specified and the error is nonfatal, program control goes to the next executable statement. The user is responsible for testing IOS and taking appropriate action. If a fatal error (one that cannot be recovered) occurs, the program will be aborted even though IOSTAT=IOS is specified.

If only ERR= was specified, program control goes to the statement specified in ERR=.

If ERR= and IOSTAT= were both specified, program control goes to the statement specified in ERR=, and status is sent to IOS.

- Entities of type BIT cannot appear as list items in input statements.

The following notes further define the function of input statements:

- A new record is read each time an input statement is executed.
- The list and format specifications determine the number of records read.
- If only a portion of a record is specified by the list and format when an input statement is executed, the rest of the record is ignored. The next input statement processes the next record.
- In the case of a formatted input statement, records are read until the list is satisfied. In either case, only enough values to satisfy the list are read.
- In the case of a formatted input statement in which the referenced format contains Hollerith or literal field descriptors, the string constant characters of the descriptors are replaced by corresponding characters from the input record.
- An unformatted input statement without a list causes one logical record of the input medium to be skipped.
- If a file is implicitly opened via an input statement, that file is opened in update mode.

Table 11-1 shows each of the possible forms of formatted and unformatted input statements.

Table 11-1 (Page 1 of 2)
Input Statements

Statement	Use
READ (u,b)list	Reads data from the file associated with the unit identifier u, formatted according to the specifications given in the format identified by b, into the items identified in the input list. Example: READ('SI',10)ARRAY(10,30),ALPHA,BETA,GAMMA
READ (u,b)	Skips one or more records on the file associated with the unit identifier u according to the number of effective slash edit descriptors in the format identified by b. Example: READ (14,45) 45 FORMAT (/,/)
READ b	Skips one or more records on the resource assigned to the logical file code 'SI'. Example: READ 45 !THIS WILL SKIP 3 RECORDS 45 FORMAT (/,/)
READ b,list	Transfers data from the resource assigned to the logical file code 'SI' into the items identified in the input list. The input data are formatted according to the specifications given in b. Example: READ 25, X, Y, (Z(I), I=1,10)
READ nl	Specifies input to the previously defined NAMELIST statement. Records are read from a logical file code 'SI' until a record is encountered that contains the NAMELIST name specified in the READ statement. Variable and array values are then written to their corresponding NAMELIST variables until a record containing the NAMELIST end code is encountered.

Table 11-1 (Page 2 of 2)
Input Statements

Statement	Use
READ (u,nl)	Specifies input to the previously defined NAMELIST statement. This form of the READ statement differs from the previous example by the explicit use of the unit identifier rather than the default value; records are being read from a specified unit. Example: READ (6,NAM1)
ACCEPT b	Skips one or more input records on the file associated with the logical file code 'UT' according to the format identified by b. Example: ACCEPT 10 10 FORMAT(/,/,/)
ACCEPT b,list	Transfers values to the items specified by the input list according to the format identified by b. The source of the values is the file associated with the logical file code 'UT'. Example: ACCEPT 10,A,B
ACCEPT nl	Specifies input to the previously defined NAMELIST statement. Input is from the file associated with the logical file code 'UT'.
READ (u)list	Reads one logical record of data from the file associated with unit identifier u and assigns successive values from the record to items identified in the list. The file must have been previously written as an unformatted file. The number of list elements cannot exceed the number of data values from the unformatted record. You must correctly type and order the variables. Example: READ (UNIT=9) B,C(1,1)

11.9.2 Output Statements

The output statements WRITE, PRINT, PUNCH, and TYPE transfer data from specified program variables to peripheral devices or internal files.

Syntax

```
WRITE ( [UNIT=] u [, [FMT=] b] [,END=c] [,ERR=d] [,REC=r] [,IOSTAT=ios] )  [list]
  or
WRITE (u,nl)
  or
PRINT b [,list]
  or
PRINT nl
  or
PUNCH b [,list]
  or
PUNCH nl
  or
TYPE b [,list]
  or
TYPE nl
```

u — A unit specifier as described in section 11.6.1

b — A format specifier; b is a statement label or array name identifying the FORMAT statement that describes the record(s) being written. This identifier can also be a character array name or character expression that contains the format information, or it can be an asterisk (*) specifying list-directed formatting.

nl — A NAMELIST name as described in section 11.9.3.

c — An end-of-file specifier; c is the statement label to which control is transferred upon encountering the end of the medium on which the file is being written.

d — An error specifier; d is the statement label to which control is transferred upon encountering an error condition in data transfer.

r — A record specifier; r is the number of the record to be written. This specifier is used exclusively for direct access input/output.

ios — An input/output status specifier; ios is an integer word variable or array element. Execution of an output statement containing this specifier causes ios to become defined with a zero if no error condition exists, with a positive integer if an error condition exists, or with a negative value if an end-of-file condition is encountered and no error condition exists. IOSTAT values are listed with execution-time diagnostics in Appendix D.

list — An input/output list.

Rules for Use

- If the optional characters UNIT= are omitted from the unit specifier, the unit identifier u must be the first item in the control information list.
- If the optional characters FMT= are omitted from the format specifier, the format identifier b must be the second item in the control information list; the unit specifier must be the first item.
- If an asterisk is used as a format identifier, the format is determined by the data (list-directed).
- Parameters that are specified with keywords can be in any order; for example, the order of END=c and ERR=d can be reversed within the parentheses.
- The following notes define END=, ERR=, and IOSTAT= processing.

I/O Completes Normally:

Control returns to the next executable statement. If IOSTAT= was specified, IOS = 0.

EOF or EOM Detected:

If neither END= nor IOSTAT= was specified, a run-time message is generated and the task is aborted.

If only IOSTAT= was specified, program control goes to the next executable statement. The user is responsible for testing IOS and taking appropriate action.

If only END= was specified, program control goes to the statement specified in END=.

If END= and IOSTAT= were both specified, program control goes to the statement specified in END=, and status is sent to IOS.

I/O Error Detected (note that the position of the file becomes indeterminate):

If neither ERR= nor IOSTAT= was specified, a run-time message is generated and the task is aborted.

If only IOSTAT= was specified and the error is nonfatal, program control goes to the next executable statement. The user is responsible for testing IOS and taking appropriate action. If a fatal error (one that cannot be recovered) occurs, the program will be aborted even though IOSTAT=IOS is specified.

If only ERR= was specified, program control goes to the statement specified in ERR=.

If ERR= and IOSTAT= were both specified, program control goes to the statement specified in ERR=, and status is sent to IOS.

- Entities of type BIT cannot appear as list items in output statements.

The following notes further define the function of output statements:

- The execution of output statements can produce several records. An unformatted output statement produces one logical record, which can consist of several physical records.
- The list and format specifications determine the number of records produced by any output statements.
- Successive data are output until the list is exhausted.
- During output to a file that specifies a fixed length record, if the data specified in the list do not fill a record, the remainder of the record is filled with blanks (if formatted) or zeros (if unformatted).
- If a file is implicitly opened via an output statement, that file is opened in update mode.

Table 11-2 shows possible forms of formatted and unformatted output statements.

Table 11-2 (Page 1 of 3)
Output Statements

Statement	Use
WRITE (u,b)list	Writes the values of the items identified in the list to the file associated with unit identifier u or an internal file. This statement converts and arranges the output data according to the specifications given in a format identified by b. Example: WRITE ('LO', 64) A, B, C
PRINT b,list	Writes the values of the items identified in the list to the resource assigned to the logical file code 'LO'. This statement converts and arranges the output data according to the format identified by b. Example: PRINT 33, OUT(3,6), W, X, Y
PRINT b	Writes the contents of any Hollerith or literal field descriptor(s) contained in the format identified by b to the resource or file assigned to the logical file code 'LO'. If neither type of field specification is found in the format, no output transfer is performed, but one or more records are written according to the number of effective slash edit descriptors in the format. Example: PRINT 44
PRINT nl	Specifies output for use with the previously defined NAMELIST statement. Output is written to the device assigned to the logical file code 'LO'. This statement results in a minimum of three records being sent to 'LO'. The first record contains the NAMELIST name; the following records contain the symbols and current values for items specified in the NAMELIST statement in order of definition. Records are written until all symbols for the NAMELIST name have been written. The NAMELIST end code is then written. Example: PRINT NAM1

Table 11-2 (Page 2 of 3)
Output Statements

Statement	Use
PUNCH b,list	Writes the values of the items identified in the list to the resource assigned to the logical file code 'BO'. This statement converts and arranges the output data according to the format identified by b. Example: PUNCH 33, OUT(3,6),W,X,Y
PUNCH b	Writes the contents of any Hollerith or literal field descriptor(s) in the format identified by b to the resource assigned to the logical file code 'BO'. If neither type of field specification is found in the format identified by b, no output transfer is performed, but one or more records are written according to the number of effective slash edit descriptors in the format. Example: PUNCH 44
PUNCH nl	Specifies output for use with the previously defined NAMELIST statement. This statement differs from PRINT nl in that output is to the logical file code 'BO'. Example: PUNCH NAM1
WRITE (u,nl)	Specifies output for use with the previously defined NAMELIST statement. This statement differs from other output statements using NAMELIST in that output is to the unit identified by u. Example: WRITE (7,NAM1)
TYPE b	Transfers the contents of Hollerith or literal field descriptors according to the format specification b. The destination of the values is the file identified by the file code 'UT'. Example: TYPE 10

Table 11-2 (Page 3 of 3)
Output Statments

Statement	Use
TYPE b,list	Transfers values of items specified by the output list according to the format specification b. The destination of the output is the logical file code UT. Example: TYPE 10,A,B
TYPE nl	Specifies output for use with the previously defined NAMELIST statement. This statement differs from other output statements using NAMELIST in that output is to logical file code 'UT'. Example: TYPE NAM1
WRITE (u)list	Writes the values of the items identified in the list into the file associated with input/output unit u. No conversion of output data is performed. The omission of the format specifier initiates unformatted (binary) output. Example: WRITE (39) TOTAL(10,10), SUM (2,5,4)

11.9.3 Input and Output Using NAMELIST

The NAMELIST statement enables the reading and writing of data by referencing a single symbolic name instead of a format specification and argument list. This symbolic name is defined in a NAMELIST statement and represents a specific set of variables.

Syntax

```
NAMELIST/x/a,b,. . .,n[/y/o,p,. . .,z] . . .
```

x,y	NAMELIST names
a,b,. . .,n o,p,. . .,z	Variable or array names that form a NAMELIST list

Rules for Use

- NAMELIST names are symbolic names.
- A NAMELIST name is enclosed in slashes (/) in a NAMELIST statement. The list of variable or array names belonging to a NAMELIST name ends with another NAMELIST name enclosed in slashes or with the end of the NAMELIST statement.
- A variable or array name can belong to one or more NAMELIST lists.
- A NAMELIST name must be defined in a NAMELIST statement before it can be referenced and the name can be defined only once.
- Once defined, a NAMELIST name can only be used in NAMELIST input and output statements.
- Input/output conversion of NAMELIST data is the same as that described in Chapter 12. However, NAMELIST input must be in the format described in this chapter.
- A NAMELIST statement that includes variable or array names in a COMMON block can only occur after the first DATA, executable or statement function statement in an individual source module.
- If NAMELIST input is to be from a file created by the editor, you must STORE (as opposed to SAVE) the file unnumbered.
- NAMELIST can not operate on unblocked files.

Example

```
NAMELIST/NAM1/Q,B,I,L,J,K/NAM2/C,J,I,L,K
```

This NAMESLIST statement defines two NAMELIST lists, NAM1 and NAM2. The variable names I, J, L, and K belong to both NAMELIST lists. Additional examples of the use of NAMELIST appear on the following pages.

11.9.3.1 Input from a User Terminal

In interactive processing, NAMELIST prompts the user with an ampersand (&) followed by the NAMELIST name. The cursor is then positioned on the next line.

If the TSM command OPTION PROMPT has been specified, the NAMELIST prompt will consist of the first three characters of the load module name. Refer to the MPX-32 Reference Manual for more information concerning the TSM command OPTION PROMPT.

The following would be displayed for a NAMELIST named NLST1. ^ represents the cursor; XXX represents the beginning of the load module name. Line numbers have been added for the purpose of explanation.

```
Line 1      &NLST1
Line 2      ^           (^ represents the cursor)
  or
Line 1      XXX^
```

In the first example, data items can be entered beginning on line 2. In the second example, data items can be entered on line 1 and on successive lines. In either example, data items can be entered in any column.

11.9.3.2 Input from Other Than a User Terminal

In batch processing, NAMELIST input must be in the following format:

- All lines must have column one blank.
- The first line of a NAMELIST input group must have an ampersand (&) in column two.
- The NAMELIST name must begin in column three of the first line of a NAMELIST input group.

For batch processing, the first line of input for a NAMELIST named NLST1 must be:

```
␣&NLST1
```

If data items are included in the same line as the NAMELIST name, the name must be followed by a blank

```
␣&NLST1␣I(2,3)=5
```

11.9.3.3 Input Data Item Formats

Data items in NAMELIST input can take three forms:

- varsubstring=constant
- arrayname=$constant_1$, $constant_2$,. . .,$constant_n$
- arrayelement=$constant_1$, $constant_2$,. . .,$constant_n$

Commas separate the data items and a comma must follow the last item in the data for each NAMELIST name. The constants in these data items can be of type integer, real, double precision, complex, logical, bit or character. If a constant is logical, it can be of the form T or .TRUE., F or .FALSE..

varsubstring=constant

varsubstring is a single variable name or a substring. Substring expressions must be integer constants. For example:

```
J=4, B=3.2, C(2:4)='ABC',
```

arrayname=$constant_n$

arrayname is the name of the array. The constants, separated by commas, are placed in the array in column-major order. The number of constants must not exceed the number of elements in the array. Successive occurrences of the same constant can be represented by k*constant, where k is the number of repetitions. For example:

```
L=2,3,7*4,
```

arrayelement=$constant_n$

arrayelement is an individual array element; that is, an array name followed by a list of integer constants within parentheses. Data item constants are separated by commas. If one constant is listed, it is placed in the specified array element. If more than one is listed, they are placed in the array, starting with the specified element and continuing in column-major order.

The number of constants must not exceed the number of array elements between the specified array element and the last array element. Successive occurrences of the same constant can be represented by k*constant, where k is the number of repetitions. For example:

```
A(4,3)=5,8,3*2,7,
```

Any subset of the set of variable or array names specified in the NAMELIST statement can be transmitted as input; i.e., the entire set need not be included. The subset is defined by including in the data items only those variable or array names to which the user wants to assign values.

The order of the data items in NAMELIST input does not have to match the order of the variable and array names in the NAMELIST statement.

Trailing blanks after integers and exponents in numeric fields are interpreted as zeros. Embedded blanks are not permitted in names or constants, except for constants of type CHARACTER. Embedded quotation marks (") or apostrophes (') are not permitted for data of type CHARACTER.

In batch processing, the end of a group of NAMELIST data items is denoted by &END. Data items following the &END are ignored for that input group.

In interactive processing, a group of NAMELIST data items can be terminated by &END or by pressing CTRL/C.

11.9.3.4 Output Data Formats

A NAMELIST output group consists of the NAMELIST name on one line followed by the data items on successive lines. A value is listed for each of the variable names specified in the associated NAMELIST statement in the same order as defined in that statement. If a data item listed in the NAMELIST statement does not have a value in the input record, it retains its previous definition. Array element values are listed in column-major order. A line containing &END is the last line of a NAMELIST output group.

Examples, Batch

The following is an example of the use of the NAMELIST statement and input in batch processing. Line and column numbers have been added for purposes of explanation.

```
REAL Q(3)
INTEGER I(3,3), L(3,3), K(4,2)
NAMELIST/NAM1/Q,B,I,L,J,K/NAM2/C,J,I,L,K
READ (5,NAM1)
  .
  .
  .
WRITE (6,NAM2)
```

The NAMELIST statement defines two NAMELIST lists, NAM1 and NAM2. The READ statement causes data to be read from the resource associated with unit specifier 5 and written to the variables and arrays specified by NAM1. The data consists of:

```
                Column 2

Line 1          ␣&NAM1␣␣I(2,3)=5,
Line 2          ␣J=4, B=3.2, Q(3)=4.0,
Line 3          ␣L=2,3,7*4,
Line 4          ␣K(3,1)=2,4,5,3*7,&END
```

Line 1 is read and examined to verify that the NAMELIST name is consistent with the name specified in the READ statement. If it is not, an error results.

When the remainder of line 1 is read, the integer constant 5 is placed in I(2,3). When line 2 is read, the integer constant 4 is placed in J; the real constants 3.2 and 4.0 are placed in B and Q(3), respectively.

When line 3 is read, the entire array is filled in column-major order with the constants listed because L is an array name not followed by a subscript. Therefore, the integer constants 2 and 3 are placed in L(1,1) and L(2,1), respectively, and the integer constant 4 is placed in L(3,1), L(1,2),. . .L(3,3).

When line 4 is read, the integer constant 2 is placed in array element K(3,1), and the remaining integer constants are placed in the remaining array elements in column-major order. Four is placed in K(4,1), 5 is placed in K(1,2), and 7 is placed in K(2,2), (3,2), and (4,2).

The WRITE statement causes data to be read from the variables and arrays specified by NAM2 and written to the resource associated with unit specifier 6. Assume that the values of J, L, I(2,3), and K(3,1) through K(4,2) were not altered since the previous READ statement; that C=428.0E+03; that I(1,3)=6; and that the rest of the elements of I and K were set to zero. The output is as follows:

```
                Column 2

Line 1          &NAM2
Line 2          C=428000.00,J=4,I=0,0,0,0,0,0,6,5,0,
Line 3          L=2,3,4,4,4,4,4,4,4,K=0,0,2,4,5,7,7,
Line 4          &END
```

Example, Interactive

The following is an example of the use of the NAMELIST statement and input from a user terminal. Line and column numbers have been added for the purpose of explanation.

```
LOGICAL L
INTEGER I(3)
CHARACTER*10 C
DATA C/'␣␣␣␣␣␣␣␣␣␣'/
NAMELIST /NLIST1/I, L, C
READ (5, NLIST1)
WRITE (6, NLIST1)
```

The NAMELIST statement defines one NAMELIST list, NLIST1. The READ statement causes data to be read from the resource associated with unit specifier 5--a user terminal--and written to the variables and arrays specified by NLIST1. The input consists of:

```
                Column 1

Line 1          &NLIST1
Line 2          I=2*1,3,C(2:4)='ABC',
Line 3          L=.TRUE.,
```

Line 1 is a prompt. Lines 2 and 3 are entered by the user; they can begin in any column. The input group is terminated by pressing CTRL/C.

The integer constant 1 is placed in I(1) and I(2); the integer constant 3 is placed in I(3); the data 'ABC' are placed in bytes 2, 3, and 4 of the variable C; the logical constant .TRUE. is placed in L.

The WRITE statement causes data to be read from the variables and arrays specified by NLIST1 and written to the resource associated with unit specifier 6, a user terminal. The output is displayed on the terminal in the following format:

```
                Column 1

Line 1          &NLIST1
Line 2          I=1,1,3,L=T,C='ßABCßßßßßß',
Line 3          &END
```

11.10 List-Directed Formatting

List-directed input/output eliminates the need to be concerned with card columns, line boundaries, and format statements. An asterisk (*) used as a format identifier in an input/output statement indicates list-directed formatting. The data to be transferred and the type of each transferred datum are specified by the contents of an inut/output list included in the input/output command.

Examples

```
WRITE(6,*)I,INK,L
WRITE(FMT=*,UNIT=6)I,(A(I),I=1,9)
READ ('SI',*)K,KAN,LIMA,M
```

A BACKSPACE statement should not be executed for a unit connected to a file that is used for list-directed input/output. If such a combination is used, the results are unpredictable. The record specifier REC=rec must not be used because list-directed input/output is record-oriented to or from a formatted sequential file, while the record specifier is used for direct access files. List-directed formatting should not operate on unblocked files.

The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a constant, a null value for input, or one of the forms

```
r*c
r*
```

c — A constant

r — A positive integer constant that specifies the number of times the constant represented by c is to be repeated. The r* form is equivalent to r successive null values.

Neither of the forms r*c or r* may contain embedded blanks, except where permitted within the constant c if c is of CHARACTER type.

A value separator is one of the following:

- A comma optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks
- A slash optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks
- One or more contiguous blanks between two constants or following the last constant

Examples

5*4	represents	4,4,4,4,4
2*'MAKE'	represents	'MAKE', 'MAKE'
7*	represents	seven null values

11.10.1 List-Directed Input

Value separators in data for list-directed input must comply with the following:

- Value separators may be preceded or followed by any number of blanks or line terminators (end-of-record, carriage return, etc.); any such combination is treated as only a single separator.
- A null value is specified by one of the following:

  Having no characters between successive value separators.

  Having no characters preceding the first value separator in the first record read by each execution of a list-directed input statement.

  The r* form. Any number of blanks may be placed between the value separators.

  Each time a null item is specified in the input data, its corresponding list item is left unchanged. A null value may not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant.

- Blanks in a list-directed input record are considered to be part of some value separator, except for the following:

  Blanks embedded in a character constant.

  Embedded blanks surrounding the real or imaginary part of a complex constant.

  Leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma.

- Slashes (/) cause the current input operation to be terminated, even if all the items of the directing list are not filled. The contents of items of the directing input/output list that have not received input before the transfer is terminated remain unchanged. If the input/output list of the controlling input/output statement has been satisfied, the use of the slash as a delimiter is optional.
- Once the input/output list has been satisfied, any items remaining in the input record are ignored.

Data for list-directed transfers are composed of alternate constants and delimiters. The constants should have the following characteristics:

- Input constants must be of an acceptable type.
- Leading blanks are allowed in the first record read by each execution of a list-directed input statement.
- Decimal points may be omitted from real and double precision constants that do not have a fractional part. The decimal point is assumed to follow the rightmost digit.
- The input form of complex data is a left parenthesis followed by the real part, a comma, and the imaginary part, followed by a right parenthesis. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.
- The input form of logical data must not include either slashes or commas among the optional characters permitted for L editing.
- The input form of character data consists of a nonempty string of characters enclosed in apostrophes. Each apostrophe (or quotation mark) within the string is represented by a pair of apostrophes (or quotation marks) without an intervening blank or end of record. Character constants may be continued from the end of one record to the beginning of the next record, for as many records as needed. The characters blank, comma, and slash may appear in these constants. Note that an embedded apostrophe pair cannot immediately follow the initial apostrophe; also, a quotation mark should be followed either by a quotation mark or a pair of hexadecimal digits denoting an ASCII character code.
- Given a character type list item of length l and a character string of length n to be transmitted to the item, the following occurs: If l is less that n, the string is truncated; that is, the leftmost characters of the string fill the item. If l is greater than n, the string is left-justified in the item and the rest of the item is blank-filled.

11.10.2 List-Directed Output

The form of the values produced by list-directed output is the same as that required for input, except as noted otherwise. With the exception of character constants, the values are separated by one of the following:

- One or more blanks
- A comma optionally preceded and/or followed by one or more blanks

New records are begun as necessary. Neither the end of a record nor blanks may occur within the constant, except in the case of complex constants and character constants.

Slashes, as value separators, and null values are not output.

Each output record begins with a blank character to provide carriage control when the record is printed.

Characteristics of output data are:

- Integer output constants are produced with the effect of an I20 edit descriptor.
- Real and double precision constants are produced with the effect of an E20.13 edit descriptor.
- Complex constants are produced with the effect of a '(', E23.16, ',', E23.16,')' edit descriptor. The end of a record may occur between the comma and imaginary part of the constant. Embedded blanks may appear between the comma and end of a record, and one blank may appear at the beginning of the next record if a complex constant is split over two lines. All listed output is preceded by leading blanks; therefore, blanks may occur between fields.
- Logical output constants are T for the value true and F for the value false in the form L20.
- Character constants produced are not delimited by apostrophes, nor are they preceded or followed by value separators. Each internal apostrophe is represented externally by one apostrophe; a blank character is inserted by the processor for carriage control at the beginning of each record that begins with a continuation of a character constant from the preceding record.

  With the exception of the blank at the beginning of the record, character constants are not preceded or followed by a value separator.

11.11 Auxiliary Input/Output Statements

Auxiliary input/output statements are used to control the connection, position, and file marking of files or devices, or to query attributes of a connection to a file. The auxiliary input/output statements OPEN, CLOSE, INQUIRE, BACKSPACE, BACKFILE, SKIPFILE, ENDFILE, and REWIND are discussed on the following pages.

11.11.1 OPEN Statement

The OPEN statement is used to connect a device or an existing file to a unit, create a file and connect it to a unit, or change certain attributes of a connection between a device or file and a unit.

Syntax

OPEN ([UNIT=] u [,specifier=item]. . .)

u — The unit identifier. It is defined in the description of the UNIT specifier on the following pages.

specifier — A keyword that identifies a particular specification.

item — An expression of a type required by the specifier.

OPEN Statement Specifiers

The OPEN statement specifiers are listed below in alphabetical order, grouped according to the mode (native or compatible) in which they can be used. The specifiers and their corresponding items are defined on the following pages in alphabetical order. The mode in which the specifier can be used is listed opposite the specifier definition; not all specifiers can be used in both modes.

Some specifiers require the use of certain other specifiers. Such cases are noted in the specifier definitions and in the Rules for Use section following the definitions.

Native Mode Specifiers

ACCESS
ALLOCATE
ALTUNIT
BLANK
BLOCKED
CLEAR
CONTIGUOUS
CONTROLBITS
DENSITY
DEVICE
ERR
EXTEND
EXTENDIBLE
FILE
FILESIZE
FORM
INCREMENT
IOSTAT
MAXSIZE
MININCREMENT
OPENMODE
OTHERACCESS
OWNERACCESS
PROJECT
PROJECTACCESS
QUEUE
RECL
REEL
SHARED
SPOOLFILE
START
STATUS
UNIT
VOLUME
WAIT

Compatible Mode Specifiers

ACCESS
ALTUNIT
BLANK
BLOCKED
CLEAR
DEVICE
ERR
FILE
FILESIZE
FORM
IOSTAT
KEY
PASSWORD
READONLY
RECL
REEL
SPOOLFILE
STATUS
UNIT
USER
VOLUME
WAIT

ACCESS=acc

native,compatible

acc is a character expression that specifies the access method for the connection of a file. The value of acc when any trailing blanks are removed is SEQUENTIAL or DIRECT. The default for acc is SEQUENTIAL. For existing files, the specified access method must be included in the set of allowed access methods for the file.

ALLOCATE=alacc

native

alacc is a character expression indicating the access mode associated with the resource when the resouce is allocated. The value of this specifier must be a subset of the set of combined modes established in the access rights specifiers OWNERACCESS, PROJECTACCESS, and OTHERACCESS. For example, 'RWMA' may be specified for alacc only if each mode was specified in at least one of the other access specifiers. Possible modes are:

'R'	Read	'A'	Append	'M'	Modify
'W'	Write	'U'	Update	' '	Use system default

Each mode may appear, at most, once in the string. Embedded blanks are not permitted. Trailing blanks are ignored. The access mode defaults to any allowable access rights for a specific file. If the ALLOCATE keyword is omitted, all allowable access modes which were established when the file was created are permitted. For example, all access modes are allowed for the owner of a file; however, update mode may not be allowed for the same file accessed under PROJECTACCESS or OTHERACCESS if this mode was not specified during file creation. If the QUEUE or SPOOLFILE is used, the system will force append mode.

ALTUNIT=alt

native,compatible

alt is a unit identifier that must be one of the forms listed for u (refer to the UNIT specifier). Use of this specifier causes all the properties that exist for the connection of the unit specified by alt to also become the properties for the connection of the unit specified by u. For example, the statement sequence:

```
OPEN (UNIT=6, FILE='FILEX', ACCESS='SEQUENTIAL',
  BLOCKED=.TRUE.)

OPEN (UNIT=7, ALTUNIT=6)
```

results in units 6 and 7 being connected to the file FILEX for blocked, sequential access. Any change of the connection between unit 6 and file FILEX from this point on has no effect on the connection between unit 7 and file FILEX.

The unit specified by the value of alt must be connected to a resource before the execution of the OPEN statement containing the specifier.

In native mode, if the ALTUNIT specifier is given in an OPEN statement, the only other specifiers allowed are UNIT, ALLOCATE, and OPENMODE. In the compatible mode, if the ALTUNIT specifier is given in an OPEN statement, the only other specifier allowed is UNIT.

BLANK=blnk

native,compatible

blnk is a character expression whose value is NULL or ZERO when any trailing blanks are removed. 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. The default for blnk is NULL; however, runtime option 8 can be used to cause the default for blnk to be ZERO. This specifier is valid only for a file being connected for formatted input/output.

BLOCKED=blk

native,compatible

blk is a logical expression. If its value is .TRUE., the file is opened as a blocked file; otherwise, it is treated as an unblocked file. The default for blk is .FALSE. (refer to the MPX-32 Reference Manual for a discussion of blocked files).

CLEAR=cl

native,compatible

cl is a logical expression. If its value is .TRUE., the file is zeroed (i.e., filled with zeroes) when created. This specifier is valid only when a file is being created. The default for cl is .FALSE.. The CLEAR option has no effect on temporary files in the compatible mode.

CONTIGUOUS=cont

native

cont is a logical expression. A value of .TRUE. indicates that extensions to an extendible file will be obtained contiguously if possible. However, if no free contiguous space exists, the extension will be obtained from any available space on the disc. A value of .FALSE. indicates the file need not be contiguous. .FALSE. is the default for cont.

CONTROLBITS=cntrl

native

cntrl is an integer expression that specifies the special control bits (bits 9-12 in word 2) in the File Control Block (FCB) the user wishes to set; e.g., parity, density, format inhibit. Refer to the MPX-32 Reference Manual for more information concerning the FCB. An integer number corresponding to the correct bit settings should be entered (e.g. cntrl=15 would set all four bits). A shift of the integer number to the correct bit fields for the control bits is done internally by the Scientific Run-Time Library and must not be done by the user.

DENSITY=den

native

den is a character expression that contains the density specification to use while connected to an XIO high speed tape unit. The tape unit must be in software select mode and at load point in order for the density to change. Possible densities are as follows:

- N indicates 800 BPI nonreturn to zero inverted (NRZI)
- P indicates 1600 BPI phase encoded (PE)
- G indicates 6250 BPI group encoded recording (GCR)

The default for den is 6250 BPI.

If DENSITY is specified, the device connected must be an XIO high speed tape unit, and a DEVICE specifier is required.

DEVICE=dev

native,compatible

dev is a character expression whose value when any trailing blanks are removed is the ASCII device mnemonic (refer to Appendix A), device channel address, and device subaddress of channel address, and device subaddress of the device that is to be connected to the specified unit. The value of dev must have the form 'ttccss' where:

- tt Two ASCII character device mnemonic
- cc Two ASCII character device channel address
- ss Two ASCII character device subaddress

For example, each of the following is acceptable:

```
DEVICE='MT0901'
DEVICE='LP7A'
```

If DEVICE is specified, FILE and ALTUNIT must be omitted and a disc device must not be specified.

Do not use the DEVICE specifier to obtain temporary disc space on a mounted volume. Use the FILE specifier for this function. If the volume on which temporary disc space is to be obtained is not important, specify STATUS=SCRATCH.

ERR=s

Native,compatible

s is the statement label of an executable statement that appears in the same program unit. If an OPEN statement contains the ERR specifier and the processor encounters an error condition during execution of the OPEN:

1. Execution of the OPEN statement terminates
2. The position of the file specified in the OPEN statement becomes indeterminate
3. Execution continues with the statement labelled s
4. If the OPEN statement contains an input/output status specifier (IOSTAT), the variable or array element ios becomes defined with a positive integer value.

EXTEND=extd

native

extd is a character expression indicating the method by which a file can be extended. Possible choices are:

'AUTO' Indicates a file will extend automatically as defined by the INCREMENT, MININCREMENT, and MAXSIZE specifiers when end-of-medium (EOM) is encountered.

'MANUAL' Indicates extensions must be handled manually by use of the extend function (X_EXTEND) when EOM is encountered.

This specifier is only valid if the EXTENDIBLE specifier is given. If EXTENDIBLE=.TRUE. and the EXTEND specifier is omitted, the file is automatically extendible.

EXTENDIBLE=extbl

native

extbl is a logical expression. If the value of extbl is .TRUE., the file is extendible; otherwise, the file is not extendible. The default is .TRUE., in which case the file is automatically extendible.

FILE=fname

compatible

fname is a one-to-eight character expression whose value when any trailing blanks are removed is the name of the file to be connected to the specified unit, or it indicates that a system file is requested when used in conjunction with the SPOOLFILE specifier. Allowable types of system files are SLO, PRINT (indicates SLO), SBO and PUNCH (indicates SBO).

FILE=fname

native

fname is a character expression representing either the pathname of the resource to which the user wishes to be connected and opened, or it indicates that a system file is requested when used in conjunction with the SPOOLFILE specifier. Allowable types of system files are SLO, SBO, PRINT (separate SLO files for each open) and PUNCH (separate SBO files for each open). To specify a temporary file on a specific volume, specify only the volume name in the pathname.

FILESIZE=fsz

native,compatible

fsz is an integer expression whose value is the size (in sectors) of the file to be created (appropriate only when creating disc files; one sector = 192 words). The value of fsz will be rounded up to the next larger allocation unit of the volume. The default for fsz is the allocation unit size for the specified disc.

FORM=fm

native,compatible

fm is a character expression whose value when any trailing blanks are removed is FORMATTED or UNFORMATTED. It specifies that a file is being connected for formatted or unformatted input/output, respectively. The default for fm is UNFORMATTED if a file is being connected for direct access and FORMATTED if a file is being connected for sequential access. For an existing file, the specified form must be the same as the form under which the file was written.

INCREMENT=inc

native

inc is an integer expression indicating the number of sectors desired with each extension of a file. If the file is automatically extendible, inc designates the extension increment each time the file is extended. If the file is manually extendible, the identifier inc will be used only if it is not specified in the manual extension function (X_EXTEND). This specifier does not apply to nonextendible files. The default for inc is 64 sectors.

IOSTAT=ios

native,compatible

ios is an integer word variable or array element that becomes defined with a zero value if no error condition exists or with a positive integer value if an error condition exists. IOSTAT values are listed with execution-time diagnostics in Appendix D.

KEY=ky

compatible

ky is a character expression containing a maximum of eight characters that specify the user key. This specifier is required if a key is associated with the user name (refer to the USER specifier).

MAXSIZE=max

native

max is an integer expression indicating the maximum size to which a file can be extended. This specifier can only be used for extendible files. Once this value is set, it is used each time automatic extension is specified. The default for max is the maximum space available on the volume.

MININCREMENT=minc

native

minc is an integer expression indicating the minimum acceptable sector size on each extension of a file. If the file is automatically extendible, minc designates the minimum acceptable extension increment each time the file is extended. This specifier does not apply to nonextendible files. The default for minc is 32 sectors.

OPENMODE=opacc

native

opacc is a character expression indicating the access mode desired when a file is opened. This parameter must contain exactly one of the modes that was specified in the ALLOCATE specifier. Possible modes are:

'R'	Read	'M'	Modify
'W'	Write	'A'	Append
'U'	Update	' '	Use system default

'W' and 'A' are invalid for direct access. If the OPENMODE keyword is omitted, update mode is the default except if the QUEUE or SPOOLFILE specifier is used. In that case, the OPENMODE defaults to append.

OTHERACCESS=otacc

native

otacc is a character expression indicating the access mode(s) associated with the resource when it is created. This specifier applies to users other than the owner of the resource and members of the owner's project. Possible modes are:

'R'	Read	'A'	Append
'W'	Write	'D'	Resource may be deleted
'U'	Update	'N'	No access
'M'	Modify	' '	Use system default

Any or all of the modes ('R','W','U','M','A','D') can be specified in a concatenated string for a new file. Each mode can appear, at most, once in the string. 'N' must not be used in a concatenated string. Embedded blanks are not permitted. Trailing blanks are ignored.

OWNERACCESS=owacc

native

owacc is a character expression indicating the access mode(s) associated with this resource when the resource is created. This specifier applies to owner name access rights only. Possible access modes are:

'R'	Read	'A'	Append
'W'	Write	'D'	Resource may be deleted
'U'	Update	'N'	No access
'M'	Modify	' '	Use system default

Any or all of the modes ('R','W','U','M','A','D') can be specified in a concatenated string for a new file. Each mode can appear, at most, once in the string. 'N' must not be used in a concatenated string. Embedded blanks are not permitted. Trailing blanks are ignored.

PASSWORD=pass

compatible

pass is the character expression that specifies a password. If a file is new, a password is assigned; if a file is old, the correct password must be specified.

PROJECT=pname

native

pname is a character expression specifying the project name that is associated with the resource.

PROJECTACCESS=pacc

native

pacc is a character expression indicating the access mode(s) associated with the resource when it is created. This specifier applies to project access rights only. Possible access modes are:

'R'	Read	'A'	Append
'W'	Write	'D'	Resource may be deleted
'U'	Update	'N'	No access
'M'	Modify	' '	Use system default

Any or all of the modes ('R','W','U','M','A','D') can be specified in a concatenated string for a new file. Each mode can appear, at most, once in the string. 'N' must not be used in a concatenated string. Embedded blanks are not permitted. Trailing blanks are ignored.

QUEUE=que

native

que is a character expression that specifies the system spoolfile queue upon which a file will be placed when the file is closed and deallocated. Possible values are:

SLO	place the file on the print spoolfile queue
SBO	place the file on the punch spoolfile queue

If this specifier is omitted, the file will not be queued. If this specifier is used, the OPENMODE will default to append due to SLO restrictions. The SPOOLFILE specifier must not be used with the QUEUE specifier.

READONLY=ro

compatible

ro is a logical expression. If ro has the value .TRUE.; the file is opened with read-only access; otherwise, read/write access is assumed.

RECL=rcl

native,compatible

rcl is an integer expression whose value must be positive. It specifies the length, in bytes, of each record in a file being connected for direct access. For an existing file, the value must match the record length; otherwise, unpredictable results will occur. For a new file, the processor creates the file with records of the specified length. RECL must be specified when a file is being connected for direct access; otherwise, it must be omitted.

REEL=rel

native,compatible

rel is a character expression with a maximum of four characters that is used for reel identification of magnetic tapes. The specified name will be used in the MOUNT message if such a message is necessary (refer to the MPX-32 Reference Manual).

SHARED=shar

native

shar is a logical expression indicating whether the file is to be shared. A value of .TRUE. indicates a file can be shared among other tasks that open the file with SHARED=.TRUE.. This is referred to as explicit sharing. A value of .FALSE. indicates that a file is being used exclusively by a task and is not shareable by any other task. If SHARED is not specified, the default is implicit sharing, which provides that one task may open the file with both READ and WRITE access while other tasks may open the file concurrently as READONLY. It is your responsibility to ensure synchronous access to shared files through the use of X:FSLR, X:FSLS, X:FXLR, X:FXLS (refer to the Scientific Run-time Library Reference Manual).

SPOOLFILE=spl

native,compatible

spl is a logical expression. If its value is .TRUE., the file to be opened is to be a spooled system file (i.e., SLO, SBO). When spl is .TRUE., the FILE specifier is used to identify the spooled file (which must be an SLO or SBO file specified as 'SLO', 'SBO', 'PRINT' or 'PUNCH') to be opened. For example,

```
FILE='SLO'
```

If the value of spl is .TRUE., and the FILE specifier is SLO or SBO, then a single SLO or SBO file will be created, regardless of the value of the STATUS specifier. If the FILE specifier is PRINT instead of SLO or PUNCH, a single SLO or SBO file will be created for compatible mode or separate SLO or SBO files will be created for native mode. When the file is closed, the SLO and SBO file is automatically linked to the print or punch queue. If this specifier is used, the access mode will default to append.

Input/output to any spooled file must be formatted, blocked, and sequential. (Refer to the MPX-32 Reference Manual for information on the use of spooled files.)

START=star

native

star is an integer expression that indicates the absolute starting block number for file creation. If the starting block number is unavailable, creation of the file will be denied. If START=0, file space is allocated wherever available.

STATUS=sta

native,compatible

sta is a character expression whose value when any trailing blanks are removed is OLD, NEW, SCRATCH, or UNKNOWN. The following rules apply to the value of sta:

- If OLD or NEW is specified, the FILE specifier must be present.
- If OLD is specified, the file must exist.
- If NEW is specified, the file must not exist.
- Successful execution of an OPEN statement with NEW specified creates the file and changes the status of the file to OLD.
- If SCRATCH is specified with an unnamed file, the file is connected to the specified unit for use by the executable program; however, the file is deleted at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program.
- SCRATCH must not be specified with a named file.
- If the value of sta is UNKNOWN and the file exists, it is opened. If the value of sta is UNKNOWN and the FILESIZE specifier is used but the file does not exist, it is created and its status changed to OLD.
- STATUS has no effect on alternate assignments or system file assignments.
- STATUS can not be used with device assignments.

The default for sta is UNKNOWN.

UNIT=u

native,compatible

u is the unit identifier. The specifier UNIT is optional; however, the unit identifier is required in all OPEN statements. An external unit identifier can be one of the following:

- An integer expression whose value must be either between 0 and 999, or a left-justified, blank-filled string of one to three ASCII characters (referred to as a logical file code) contained in an INTEGER*4 variable.
- A logical file code constant in the form 's', where s is a string on one to three characters.

USER=usr

compatible

usr is a character expression with a maximum of eight characters representing the user name associated with the file to be opened. For example, the following statement creates a file named NOTES under the user name JOHN.

```
OPEN(UNIT=1, STATUS='NEW', USER='JOHN', FILE='NOTES')
```

If the USER specifier is not in the OPEN statement, the user name defaults to the name associated with the executing program.

A blank user name, i.e., USER=' ', is used to specify a system file. (Refer to the MPX-32 Reference Manual for a discussion of user name attributes).

VOLUME=vol native,compatible	vol is an integer expression that contains a volume number for a magnetic tape. Supplying vol in an OPEN statement associates the integer value with the currently mounted volume and indicates multivolume magnetic tape processing. When an end-of-medium is encountered, a mount message is issued for the next volume, if required. The default for magnetic tapes is single volume only. This specifier must not be used for nontape devices.
WAIT=w native,compatible	w is a logical expression. If the value of w is .TRUE., the processor will wait until the desired resource is free and can be allocated; otherwise, the processor will take a denial return immediately if the resource cannot be allocated. The default for this argument is .TRUE..

Rules for Use

- The UNIT parameter can be used without the keyword UNIT. However, keywords must be used with all other parameters.
- When the UNIT parameter is used without its keyword, it must be the first parameter. Otherwise, UNIT and the other parameters can be in any order.
- No specifier can appear in the same OPEN statement more than once.
- A comma must precede the first specifier following the unit identifier (u), and a comma must precede any additional specifiers.
- An equal sign must separate a specifier and its corresponding item.
- The following types of OPEN statements require the following specifiers in addition to the unit specifier:

  Sequential access to permanent disc file assignments: FILE

  Direct access to permanent disc file assignments: ACCESS, FILE, RECL

  Sequential access to spooled file assignments: BLOCKED, SPOOLFILE, FILE

  Sequential access to temporary disc file assignments: none

  Direct access to temporary disc file assignments: ACCESS, RECL

  Sequential access to device assignments: DEVICE

  Direct access to device assignments: DEVICE, ACCESS, RECL

  Association of a LFC with another file: ALTUNIT

- If the value of STATUS is OLD or NEW, the FILE specifier is required.
- The RECL specifier is required for all direct access OPEN statements.
- A unit can be connected by the execution of an OPEN statement located within any program unit of an executable program. Once an external unit is connected, it may be referenced in any program unit of the executable program.

- If an OPEN is attempted on a currently open logical file code (LFC) and the parameters specified for the new OPEN match those already in effect, then the new OPEN attempt will be ignored, no error status will be returned, and no parameters will be changed for the current open state. This is also true if only parameters from the following list differ in the new OPEN. They are: BLANK, CLEAR, CONTIGUOUS, ERR, EXTEND, EXTENDIBLE, INCREMENT, IOSTAT, MAXSIZE, MININCREMENT, OTHERACCESS, OWNERACCESS, PROJECTACCESS, START, STATUS, and WAIT. If any of the parameters other than those stated above differ from the parameters in effect, the current connection to the LFC will be closed then deallocated and the new OPEN will be attempted with normal error reporting procedures in effect should an error occur.

- If a file is connected to a unit, execution of an OPEN statement on that file and a different unit is not permitted; i.e., connecting two units to the same file. The ALTUNIT specifier provides the means for connecting two units to the same file.

- If a DENSITY specifier is used, the DEVICE specifier is required, and the device must be an XIO high speed tape unit.

Examples

```
OPEN (UNIT=10, FILE='NOTES')
```

The preceding statement causes a file named 'NOTES' to be connected to unit number 10.

```
OPEN (UNIT=6, FILE='LISTING', QUEUE='SLO', ALLOCATE='A', OPENMODE='A')
```

The preceding statement causes a file named 'LISTING', to be connected to unit number 6. The file is to be allocated in the append mode; therefore, it must be opened in the append mode as indicated by OPENMODE='A'. When the file is closed, it will be placed on the print spoolfile queue (SLO).

11.11.2 CLOSE Statement

The CLOSE statement is used to terminate the connection of a particular file to a unit.

Syntax

CLOSE ([UNIT=] u [,specifier=item]. . .)

u — The unit identifier. It is defined in the description of the UNIT specifier in the previous section.

specifier — A keyword that identifies a particular specification. CLOSE statement specifiers and their corresponding items are listed below.

item — An expression of a type required by the specifiers.

ERR=s — ERR is an error specifier; s is the statement label of an executable statement in the same program unit as the error specifier. If a CLOSE statement contains an error specifier and the processor encounters an error condition during execution of the CLOSE:

1. Execution of the CLOSE statement terminates
2. The position of the file specified in the CLOSE statement becomes indeterminate
3. Execution continues with the statement label s
4. If the CLOSE statement contains an input/output status specifier (IOSTAT=ios), the variable or array element ios becomes defined with a positive integer value.

IOSTAT=ios — IOSTAT is an input/output status specifier; ios is an integer word variable that becomes defined with a zero value if no error condition exists or with a positive integer value if an error condition exists. IOSTAT values are listed with execution-time diagnostics in Appendix D.

STATUS=sta — sta is a character expression whose value is KEEP or DELETE when any trailing blanks are removed. This specifier determines the disposition of the file connected to the specified unit. KEEP must not be specified for a file whose status prior to execution of the CLOSE statement is SCRATCH. If KEEP is specified for a file that exists, the file continues to exist after the execution of the CLOSE statement. If DELETE is specified, the file will not exist after execution of the CLOSE statement. The default for this specifier is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH, in which case the default is DELETE. STATUS has no effect on device assignments or system file assignments.

If other units are connected to the same file, the following applies: If the value of sta is DELETE and other units are still connected to the same file, the specified unit 'u' will be closed and the file will be marked for deletion; further allocations will not be allowed for the file. When the last unit connected to the file is closed, the file will be deleted.

Rules for Use

- The UNIT parameter can be used without the keyword UNIT. However, keywords must be used with all other parameters.
- When the UNIT parameter is used without its keyword, it must be the first parameter. Otherwise, UNIT and the other parameters can be in any order.
- A comma must precede the first specifier following the unit identifier (u), and a comma must precede any additional specifiers.
- An equal sign must separate a specifier and its corresponding item.
- A CLOSE statement that refers to a unit can be executed in any program unit of an executable program. The CLOSE statement does not have to occur in the same program unit as the OPEN statement referring to that unit.
- Execution of a CLOSE statement specifying a unit that does not exist or has no file connected to it is permitted and affects no files.
- After a unit/file has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program, either to the same unit/file or a different unit/file.
- After an executable program is terminated for reasons other than an error condition, all connected units are closed. Each unit is closed with the status KEEP unless the file status prior to termination of execution was SCRATCH, in which case the unit is closed with status DELETE.

Examples

```
CLOSE (UNIT=6, ERR=20, IOSTAT=INP)
```

The preceding statement disconnects the file connected to unit 6. Control passes to the statement labelled 20 upon detection of an error condition, and the integer variable INP becomes defined with a positive integer value. If no error condition exists INP becomes defined with a zero.

```
CLOSE (UNIT=6, STATUS='DELETE')
```

The preceding statement disconnects the file connected to unit 6 and deletes the file.

11.11.3 INQUIRE Statement

The INQUIRE statement is used to determine the attributes of a particular named file or the attributes of the connection to a particular unit. There are two forms of the INQUIRE statement: INQUIRE by file and INQUIRE by unit. All value assignments are performed according to the rules for assignment statements.

The INQUIRE statement may be executed whether or not a file is connected to a unit. All values assigned by the INQUIRE statement are those that are current at the time the statement is executed.

A variable or array name or any associated entity that is used as a specifier in an INQUIRE statement must not be referenced by any other specifier in the same INQUIRE statement.

11.11.3.1 INQUIRE by File, Native Mode

The INQUIRE by file statement examines the properties of a file or a connection.

Syntax

INQUIRE (FILE=fname [,PROJECT=pname] [,specifier=item]. . .)

fname — A character expression representing the pathname of the resource about which the user wishes to inquire. The named file need not exist or be connected to a unit. The value of fname must be of a form acceptable as a pathname.

pname — A character array element or character expression that specifies the project associated with the file. pname must be given if the project is different from the project name associated with the program in execution.

specifier — A keyword that identifies a particular specification. INQUIRE statement specifiers are listed on the pages that follow.

item — An expression of a type required by the specifier.

11.11.3.2 INQUIRE by File, Compatible Mode

The INQUIRE by file statement examines the properties of a file or a connection.

Syntax

INQUIRE (FILE=fin [,USER=usr] [,specifier=item]. . .)

fin — A character expression whose value, when any trailing blanks are removed, is the name of the permanent disc file being inquired about. The named file need not exist or be connected to a unit. The value of fin must be of a form acceptable to the processor as a file name.

usr — A character expression with a maximum of eight characters. In an INQUIRE by file, a username must be supplied if different from the username associated with the program in execution. A blank username (USER=$\emptyset$) specifies a system file. Refer to the MPX-32 Reference Manual for a discussion of username attributes.

specifier — A keyword that identifies a particular specification. INQUIRE statement specifiers are listed on the pages that follow.

item — An expression of a type required by the specifier.

11.11.3.3 INQUIRE by Unit, Native Mode

The INQUIRE by unit statement examines the properties of an existing file on a specified unit.

Syntax

```
INQUIRE ( [UNIT=] u [,PROJECT= pname] [,ALTUNIT=alt] [,VOLUME=vol]
  [,specifier=item]. . .)
```

u — The unit identifier which can be one of the following:

- An integer expression whose value must be either between 0 and 999, or a left-justified blank-filled string of one to three ASCII characters (referred to as a logical file code), contained in an INTEGER*4 variable.
- A logical file code constant in the form 's', where s is a string of one to three characters.

pname — A character variable or array element that returns the project name associated with a file when it was created.

alt — An integer variable or array element that returns the value of the unit (logical file code) that is connected to the same file to which unit u is connected. The logical file code returned is the one that was specified in the ALTUNIT specifier in the OPEN statement for unit u. A value of zero indicates that no alternate unit exists.

vol — An integer variable or array element that returns the volume number of the currently mounted volume for a magnetic tape device.

specifier — A keyword that identifies a particular specification. INQUIRE statement specifiers are listed on the pages that follow.

item — An expression of a type required by the specifier.

11.11.3.4 INQUIRE by Unit, Compatible Mode

The INQUIRE by unit statement examines properties of an existing file on a specified unit.

Syntax

```
INQUIRE ( [UNIT=] u [,USER=usr] [,ALTUNIT=alt] [,VOLUME=vol]
[,specifier=item]...)
```

u — The unit identifier which can be one of the following:

- An integer expression whose value must be either between 0 and 999, or a left-justified blank-filled string of one to three ASCII characters (referred to as a logical file code) contained in an INTEGER*4 variable.
- A logical file code constant in the form 's', where s is a string of one to three characters.

usr — A character variable, substring, or array element with a maximum of eight characters. In an INQUIRE by unit, this argument returns the value of the username associated with the logical file code u.

alt — An integer variable or array element that returns the value of the unit (logical file code) that is connected to the same file to which unit u. The logical file code returned is the one that was specified in the ALTUNIT specifier in the OPEN statement for unit u. A value of zero indicates that no alternate unit exists.

vol — An integer variable or array element that returns the volume number of the current mounted volume for a magnetic tape device.

specifier — A keyword that identifies a particular specification.

item — An expression of a type required by the specifier.

11.11.3.5 INQUIRE Statement Specifiers

The INQUIRE statement specifiers are listed below in alphabetical order, grouped according to the mode (native or compatible) in which they can be used. The specifiers and their corresponding items are defined on the following pages in alphabetical order. The mode in which the specifier can be used is listed opposite the specifier definition; not all specifiers can be used in both modes.

Some specifiers require the use of certain other specifiers. Such cases are noted in the specifier definitions and in the Rules for Use section following the definitions.

Native Mode Specifiers

ACCESS	FORM	OTHERACCESS
ALLOCATE	FORMATTED	OWNERACCESS
BLANK	INCREMENT	PROJECTACCESS
BLOCKED	IOSTAT	QUEUE
CONTIGUOUS	MAXSIZE	QUEUED
DEVICE	MININCREMENT	RECL
DIRECT	NAME	SEQUENTIAL
ERR	NAMED	SHARED
EXIST	NEXTREC	SPOOLFILE
EXTEND	NUMBER	UNFORMATTED
EXTENDIBLE	OPENED	
FILESIZE	OPENMODE	

Compatible Mode Specifiers

ACCESS	FILESIZE	NUMBER
BLANK	FORM	OPENED
BLOCKED	FORMATTED	READONLY
DEVICE	IOSTAT	RECL
DIRECT	NAME	SEQUENTIAL
ERR	NAMED	SPOOLFILE
EXIST	NEXTREC	UNFORMATTED

ACCESS=acc native,compatible	acc is a character variable, character substring, or character array element that returns the value SEQUENTIAL if a file is connected for sequential access or DIRECT if it is connected for direct access.
ALLOCATE=alacc native	alacc is a character variable, character substring, or character array element that returns the value of an eight character, left-justified, blank-filled, concatenated string indicating the access modes that were assigned to a file when it became connected. The file must be explicitly opened within the program before alacc is defined.
BLANK=blnk native,compatible	blnk is a character variable, character substring, or character array element that returns the value NULL or ZERO according to the type of blank control that is in effect for a file. If there is no connection, or if the connection is not for formatted input/output, blnk becomes undefined.
BLOCKED=blk native,compatible	blk is a logical variable or logical array element. It returns the value .FALSE. if a file is unblocked; otherwise, the value .TRUE. is returned for blocked files. (Refer to the MPX-32 Reference Manual for a discussion of blocked files.)

CONTIGUOUS=cont

native

cont is a logical variable or logical array element that returns the value .TRUE. if a file was marked contiguous when it was created; otherwise, .FALSE. is returned.

DEVICE=dev

native,compatible

dev is a character variable, character substring, or character array element that returns the device mnemonic (refer to section A.5), channel, and subaddress of the device to which the unit is connected.

The value of dev must have the form 'ttccss' where:

tt The two ASCII character device mnemonic

cc The two ASCII character device channel address

ss The two ASCII character device subaddress

DIRECT=dir

native,compatible

dir is a character variable, character substring, or character array element that returns the value YES if direct is included in the set of allowed access methods for a file, NO if direct is not included in the set of allowed access methods for a file, and UNKNOWN if the allowed access methods are unknown.

ERR=s

native,compatible

s is the statement label of an executable statement in the same program unit. If the processor encounters an error condition during execution of the INQUIRE statement, all inquiry specifier variables or array elements are undefined except for the variable or array element ios in any IOSTAT specifier, which becomes defined as a positive integer value. Execution continues with the statement labelled s.

EXIST=ex

native,compatible

ex is a logical variable or logical array element. Execution of an INQUIRE by file statement causes ex to return the value .TRUE. only if the file specified exists in the specified directory searched; otherwise, ex returns the value .FALSE.. Execution of an INQUIRE by unit statement causes ex to return the value .TRUE. if the specified unit exists (i.e., the set of units that exist are those in the range 0-999 with 1-3 ASCII characters, left-justified and blank-filled); otherwise, ex returns the value .FALSE..

EXTEND=extd

native

extd is a character variable, character substring, or character array element that returns the value 'AUTO' if a file is automatically extendible and 'MANUAL' if a file is manually extendible. If the file is not extendible, extd becomes undefined.

EXTENDIBLE=extbl native	extbl is a logical variable or logical array element. It returns the value .TRUE. if the file is extendible; otherwise, the value .FALSE. is returned.
FILESIZE = fsz native,compatible	fsz is an integer variable or integer array element that returns the current size (in sectors for MPX-32) of the file.
FORM=fm native,compatible	fm is a character variable, character substring, or character array element that returns the value FORMATTED if a file is connected for formatted input/output; or the value UNFORMATTED if a file is connected for unformatted input/output. If there is no connection, fm becomes undefined.
FORMATTED=fmt native,compatible	fmt is a character variable, character substring, or character array element that returns the value YES if formatted is included in the set of allowed forms for a file, NO if formatted is not included in the set of allowed forms for a file, and UNKNOWN if the allowed forms are unknown (e.g., the file is not open).
INCREMENT=inc native	inc is an integer variable or integer array element that returns the extension increment value specified for a file when the file was created. If the file is not extendible, inc becomes undefined.
IOSTAT=ios native,compatible	ios is an integer word variable or array element that returns a zero if no error condition exists or a positive integer value if an error condition exists. IOSTAT values are listed with execution-time diagnostics in Appendix D.
MAXSIZE=max native	max is an integer variable or integer array element that returns the value specified by MAXSIZE when the file was created. If a file is not extendible, max becomes undefined.
MININCREMENT=minc native	minc is an integer variable or integer array element that returns the value specified by MININCREMENT when the file was created. If a file is not extendible, minc becomes undefined.
NAME=fn native,compatible	fn is a character variable, character substring, or character array element that returns the name (pathname) of a file, if the file has a name; otherwise, it becomes undefined.
NAMED=nmd native,compatible	nmd is a logical variable or logical array element that returns the value .TRUE. if the specified file has a name; .FALSE. if it does not.

NEXTREC=nr

native,compatible

nr is an integer variable or integer array element that returns the value n+1, where n is the record number of the last record read or written on a file connected for direct access. If a file is connected but no records have been read or written since the connection, nr returns the value one. If a file is not connected for direct access, or if the position of the file is indeterminate because of a previous error condition, nr becomes undefined.

NUMBER=num

native,compatible

num is an integer variable or integer array element that returns the value of the external unit identifier (logical file code) of the unit currently connected to the file. This value consists of from one to three left-justified ASCII characters. If there is no unit connected to the file, num becomes undefined.

OPENED=od

native,compatible

od is a logical variable or logical array element. Execution of an INQUIRE by file statement causes od to return the value .TRUE. if the specified file is connected to a unit; .FALSE. if it is not. Execution of an INQUIRE by unit statement causes od to return the value .TRUE. if the specified unit is connected to a file; .FALSE. if it is not.

OPENMODE=opacc

native

opacc is a character variable, character substring, or character array element that returns the access mode that is currently in effect for an opened file or unit. If the file or unit is not open, the value of opacc becomes undefined.

OTHERACCESS=otacc

native

otacc is a character variable, character substring, or character array element that returns the value of an eight character, left-justified, blank-filled, concatenated string indicating the OTHERACCESS modes that exist for a file.

OWNERACCESS=owacc

native

owacc is a character variable, character substring, or character array element that returns the value of an eight character, left-justified, blank-filled, concatenated string indicating the OWNERACCESS modes that exist for a file.

PROJECTACCESS=pacc

native

pacc is a character variable, character substring, or character array element that returns the value of an eight character, left-justified, blank-filled, concatenated string indicating the PROJECTACCESS modes that exist for a file.

QUEUE=que native	que is a character variable, character substring, or character array element that returns the value 'SLO' if a file will be placed on the print spoolfile queue when the file is closed and deallocated. It is assigned the value 'SBO' if a file will be placed on the punch spoolfile queue when the file is closed and deallocated. If there is no connection, que becomes undefined.
QUEUED=qued native	qued is a logical variable or logical array element that returns the value .TRUE. if the file will be placed on the system spoolfile queue (SLO,SBO) when the file is closed and deallocated; otherwise, .FALSE. is returned.
READONLY=ro compatible	ro is a logical variable or logical array element that returns the value .TRUE. if the file has read-only access; otherwise, .FALSE. is returned.
RECL=rcl native,compatible	rcl is an integer variable or array element that returns the length, in bytes, of each record in a file connected for direct access. If there is no connection or if the connection is not for direct access, rcl becomes undefined.
SEQUENTIAL=seq native,compatible	seq is a character variable, character substring, or character array element that returns the value YES if sequential is included in the set of allowed access methods for a file, NO if sequential is not included in the set of allowed access methods for a file, or UNKNOWN if the allowed access methods are unknown.
SHARED=shar native	shar is a logical variable or logical array element that returns the value .TRUE. if a file is connected for shared use; otherwise, .FALSE. is returned.
SPOOLFILE=spl native,compatible	spl is a logical variable or logical array element that returns the value .TRUE. if a file is a system spoolfile (SLO,SBO). The spoolfile name will be assigned to variable fn of the NAME= specifier, if present. The value .FALSE. is returned if a file is not a system spoolfile.
UNFORMATTED=unf native,compatible	unf is a logical variable or logical array element that returns the value YES if unformatted is included in the set of allowed forms for a file, NO if unformatted is not included in the set of allowed forms for a file, or UNKNOWN if the forms of a file are unknown.

Rules for Use

- The UNIT parameter can be used without the keyword UNIT. However, keywords must be used with all other parameters.
- When the UNIT parameter is used without its keyword, it must be the first parameter. Otherwise, UNIT and the other parameters can be in any order.
- A comma must precede the first specifier following the unit identifier (u), and a comma must precede any additional specifiers.
- An equal sign must separate a specifier and its corresponding item.
- In the INQUIRE by file statement, UNIT must not appear in the specifier list.
- In the INQUIRE by file statement, the following specifiers are defined when the value of the FILE specifier is a valid file name and when the file name exists (i.e., EXIST=.TRUE. and OPENED=.FALSE.).

In Native Mode		In Compatible Mode
CONTIGUOUS	INCREMENT	NAME
DEVICE	MAXSIZE	NAMED
EXTEND	MININCREMENT	
EXTENDIBLE	NAME	
FILESIZE	NAMED	

- The following specifiers are defined when the value of the FILE specifier is a valid file name, the file name exists (i.e., EXIST=.TRUE.), and the OPENED specifier has a value of .TRUE.:

In Native Mode		In Compatible Mode	
ACCESS	NAMED	ACCESS	NAMED
ALLOCATE	NEXTREC	ALTUNIT	NEXTREC
ALTUNIT	NUMBER	BLANK	NUMBER
BLANK	OPENMODE	BLOCKED	RECL
BLOCKED	OTHERACCESS	DEVICE	SEQUENTIAL
CONTIGUOUS	OWNERACCESS	DIRECT	SPOOLFILE
DEVICE	PROJECTACCESS	FILESIZE	UNFORMATTED
DIRECT	QUEUE	FORM	USER
EXTEND	QUEUED	FORMATTED	VOLUME
EXTENDIBLE	RECL	NAME	
FILESIZE	SEQUENTIAL		
FORM	SHARED		
FORMATTED	SPOOLFILE		
INCREMENT	UNFORMATTED		
MAXSIZE	VOLUME		
MININCREMENT			
NAME			

- In the INQUIRE by unit statement, the following specifiers can be assigned values only when the specified unit exists and when a file is connected to the unit; otherwise, they are undefined.

In Native Mode

ACCESS	MAXSIZE
ALLOCATE	MININCREMENT
ALTUNIT	NAME
BLANK	NAMED
BLOCKED	NEXTREC
CONTIGUOUS	NUMBER
DEVICE	PROJECT
DIRECT	PROJECTACCESS
EXTEND	RECL
EXTENDIBLE	SEQUENTIAL
FILESIZE	SPOOLFILE
FORM	UNFORMATTED
FORMATTED	VOLUME
INCREMENT	

In Compatible Mode

ACCESS	NAMED
ALTUNIT	NEXTREC
BLANK	NUMBER
BLOCKED	RECL
DEVICE	SEQUENTIAL
DIRECT	SPOOLFILE
FILESIZE	UNFORMATTED
FORM	USER
FORMATTED	VOLUME

- If an error condition occurs during the execution of an INQUIRE statement, all of the inquiry specifiers except IOSTAT are undefined.
- The specifiers EXIST and OPEN always are defined unless an error condition occurs.
- For files preconnected by static assignments, the following specifiers are defined; all others are undefined.

ALTUNIT	IOSTAT
DEVICE	NAME
ERR	NAMED
EXIST	

Examples

```
CHARACTER*10  FAMT, SQT
INQUIRE (FILE='WHO', FORMATTED=FAMT, SEQUENTIAL=SQT, NUMBER=NUM)
```

The preceding statement is an example of an INQUIRE by file.

The following statement causes an inquiry about the name of the file, if any, connected to unit 6.

```
LOGICAL ISIT
CHARACTER*8 N
INQUIRE (UNIT=6, NAME=N, NAMED=ISIT)
```

11.11.4 BACKSPACE Statement

The BACKSPACE statement causes the file assigned to the specified unit to be backspaced one logical record. If there is no preceding record, the position of the file is unchanged.

Syntax

BACKSPACE ([UNIT=] u[,alist])

or

BACKSPACE u

u — The unit specifier as described in section 11.6.1.

alist — A list of specifiers as follows:

ERR=s — ERR is an error specifier; s is the statement label of an executable statement in the same program unit as the error specifier. If a BACKSPACE statement contains an error specifier and the processor encounters an error condition during execution of the statement:

1. Execution of the BACKSPACE statement terminates
2. The position of the file specified in the BACKSPACE statement becomes indeterminate
3. Execution continues with the statement label s
4. If the BACKSPACE statement contains an input/output status specifier (IOSTAT=ios), the variable or array element ios becomes defined with a positive integer value.

IOSTAT=ios — ios is an integer word variable that returns zero if no error condition exists or a positive integer value if an error condition exists. IOSTAT values are listed with execution-time diagnostics in Appendix D.

Rules for Use

- Exactly one unit specifier must appear in the statement.
- One of each of the alist specifiers may appear in the statement.
- Backspacing a file that is connected but does not contain any data is prohibited.
- Backspacing over records written using list-directed formatting or a NAMELIST write is prohibited.

- If an error exists, but neither the ERR= specifier nor the IOSTAT= specifier is present, the program will be aborted and a run-time error message will be written. If only the IOSTAT= specifier is present, execution of the statement terminates, the position of the file becomes indeterminate, and control proceeds to the next executable statement.

Examples

The following statement causes unit 8 to backspace 1 record.

```
BACKSPACE 8
```

The following statements cause unit 8 to backspace 10 records.

```
   DO 2 I=1,10
2  BACKSPACE (UNIT=8, IOSTAT=INP)
```

11.11.5 BACKFILE Statement

The BACKFILE statement causes the device assigned to the specified unit to be positioned to the end-of-file mark in the preceding file. If there is no preceding file, the device is positioned to the beginning-of-medium (BOM).

Syntax

```
BACKFILE ( [UNIT=] u [,alist] )

        or

BACKFILE u
```

u — The unit specifier as described in section 11.6.1.

alist — A list of specifiers as follows:

ERR=s — ERR is an error specifier; s is the statement label of an executable statement in the same program unit as the error specifier. If a BACKFILE statement contains an error specifier, and the processor encounters an error condition during execution of the statement

1. Execution of the BACKFILE statement terminates
2. The position of the file specified in the BACKFILE statement becomes indeterminate
3. Execution continues with the statement label s
4. The status specifier ios, if present, becomes defined with a positive value.

IOSTAT=ios — IOSTAT is an input/output status specifier; ios is an integer word variable that returns zero if no error condition exists, a positive value if an error condition exists, or a negative value if an end-of-file condition is encountered and no error condition exists. IOSTAT values are listed with execution-time diagnostics in Appendix D.

Rules for Use

- Exactly one unit specifier must appear in the BACKFILE statement.
- One of each of the alist specifiers may appear in the statement.
- The use of BACKFILE for a file that is connected but does not contain any data is prohibited.
- If an error exists, but neither the ERR= specifier nor the IOSTAT= specifier is present, the program will be aborted and a run-time error message will be written. If only the IOSTAT= specifier is present, execution of the output statement terminates, the position of the file becomes indeterminate, and control proceeds to the next executable statement.

11.11.6 SKIPFILE Statement

The SKIPFILE statement causes the device assigned to the specified unit to skip one file; the system's pointer will be pointing to the beginning of the succeeding file and an end-of-file bit is set. If there are no succeeding files, the device is positioned to the end-of-medium (EOM). The next READ will result in an end-of-file indicator unless CALL STATUS is issued to reset the end-of-file bit.

Syntax

```
SKIPFILE ( [UNIT=] u [,alist] )

        or

SKIPFILE u
```

u — The unit specifier as described in section 11.6.1.

alist — A list of specifiers as follows:

ERR=s — ERR is an error specifier; s is the statement label of an executable statement that appears in the same program unit as the error specifier. If a SKIPFILE statement contains an error specifier and the processor encounters an error condition during execution of the statement

1. Execution of the SKIPFILE statement terminates
2. The position of the file specified in the SKIPFILE statement becomes indeterminate

3. Execution continues with the statement label s

4. The status specifier ios, if present, becomes defined with a positive value.

IOSTAT=ios — IOSTAT is an input/output status specifier; ios is an integer word variable that returns zero if no error condition exists, a positive value if an error condition exists, or a negative value if an end-of-file condition is encountered and no error condition exists. IOSTAT values are listed with execution-time diagnostics in Appendix D.

Rules for Use

- Exactly one unit specifier must appear in the SKIPFILE statement.
- One of each of the alist specifiers may appear in the statement.
- The use of SKIPFILE for a file that is connected but does not contain any data is prohibited.
- If an error exists, but neither the ERR= specifier nor the IOSTAT= specifier is present, the program will be aborted and a run-time error message will be written. If only the IOSTAT= specifier is present, execution of the output statement terminates, the position of the file becomes indeterminate, and control proceeds to the next executable statement.

11.11.7 ENDFILE Statement

The ENDFILE statement causes an end-of-file mark to be written on the specified file or device. For formatted/unblocked magnetic tape or disc files, a data pattern of X'0F' will be written to the first byte of the end-of-file record if the default method of end-of-file processing is used. A pattern of X'0FE0FE0F' will be written to the first word of the end-of-file record, if X:MPXEOF was called at the beginning of the user's task.

Syntax

```
ENDFILE ( [UNIT=] u [,alist] )

        or

ENDFILE u
```

u — The unit specifier as described in section 11.6.1.

alist — A list of specifiers as described below.

ERR= s — ERR is an error specifier; s is the statement label of an executable statement in the same program unit as the error specifier. If an ENDFILE statement contains an error specifier and the processor encounters an error condition during execution of the statement

1. Execution of the ENDFILE statement terminates
2. The position of the file specified in the ENDFILE statement becomes indeterminate
3. Execution continues with the statement label s
4. If the ENDFILE statement contains an input/output status specifier (IOSTAT=), the variable or array element ios becomes defined with a positive integer value.

IOSTAT= ios — IOSTAT is an input/output status specifier; ios is an integer word variable that returns zero if no error condition exists or a positive integer value if an error condition exists. IOSTAT values are listed with execution-time diagnostics in Appendix D.

Rules for Use

- Exactly one unit specifier must appear in the statement.
- One of each of the alist specifiers may appear in the statement.
- If an error exists, but neither the ERR= specifier nor the IOSTAT= specifier is present, the program will be aborted and a run-time error message will be written. If only the IOSTAT= specifier is present, execution of the output statement terminates, the position of the file becomes indeterminate, and control proceeds to the next executable statement.

Examples

```
ENDFILE 8

ENDFILE (UNIT=6, IOSTAT=INP, ERR=60)
```

11.11.8 REWIND Statement

The REWIND statement causes the specified file or device to be repositioned to the beginning of the medium. If the file is already at the beginning of the medium, execution of the statement has no effect.

Syntax

```
REWIND ( [UNIT=] u [,alist] )

    or

REWIND u
```

u — The unit specifier as described in section 11.6.1.

alist — A list of specifiers as follows:

ERR= s — ERR is an error specifier; s is the statement label of an executable statement in the same program unit as the error specifier. If a REWIND statement contains an error specifier and the processor encounters an error condition during execution of the statement

1. Execution of the REWIND statement terminates
2. The position of the file specified in the REWIND statement becomes indeterminate
3. Execution continues with the statement label s
4. If the REWIND statement contains an input/output status specifier (IOSTAT=), the variable or array element ios becomes defined with a positive integer value.

IOSTAT= ios — IOSTAT is an input/output status specifier; ios is an integer word variable that returns zero if no error condition exists or a positive integer value if an error condition exists. IOSTAT values are listed with execution-time diagnostics in Appendix D.

Rules for Use

- Exactly one unit specifier must appear in the statement.
- One of each of the alist specifiers may appear in the statement.
- If an error exists, but neither the ERR= specifier nor the IOSTAT= specifier is present, the program will be aborted and a run-time error message will be written. If only the IOSTAT= specifier is present, execution of the statement terminates, the position of the file becomes indeterminate, and control proceeds to the next executable statement.

Examples

```
REWIND 8

REWIND (UNIT=10, IOSTAT=IST, ERR=30)
```

11.12 ENCODE and DECODE Statements

The ENCODE/DECODE statements provide a means of memory-to-memory data conversion between the input/output list and a user-specified internal record.

ENCODE is similar to WRITE and transfers the list elements into the internal record using specified format conversions. DECODE is similar to READ and transfers the contents of the internal record into the list elements using specified format conversions.

Note: Conversion can also be done by using internal files with WRITE/READ statements.

Syntax

```
ENCODE (n, f, v) [list]

DECODE (n, f, v) [list]
```

n — The number of characters in the internal record.

f — The statement number, variable name, array name, array element, or format statement number representing the FORMAT statement controlling the conversion.

v — The variable name, array name, or array element that is the starting location of the internal record. The name may be of any data type.

list — An input/output list (refer to 11.7).

Rules for Use

- The internal record begins with the leftmost character position of the entity specified by v and continues for n ASCII characters.
- If more than n characters are transferred, an error message will occur at execution time in the diagnostic listing. If fewer than n characters are transferred, the remainder of the internal record will be blank-filled by ENCODE or ignored by DECODE.
- ENCODE and DECODE contain no concept of multiple records; therefore, a slash (/) in the format statement is an undefined operation.

Examples

In the following example, parameters have been defined as indicated. Note that CHARACTER type may be used as the internal record entity.

```
DIMENSION A(2), B(2), ALPHA(4), A6(2), B6(2), GAMMA(4)
CHARACTER *10  ALPHA, A, B, GAMMA, A6, B6
```

A(1) contains 'ABCDEFGHIJ'
A(2) contains 'KLMNOØ9999'
B(1) contains 'PQRSTUVWXY'
B(2) contains 'Z12345TTTT'

The following statement transfers a total of 20 characters from list elements A and B into an internal record whose starting location is denoted by ALPHA.

```
   ENCODE (20,1,ALPHA) A,B
1  FORMAT (A8, A2, A7, A3)
```

Characters are transferred according to the format specified by statement number 1 (eight characters from A(1), two from A(2), seven from B(1), and three from B(2)).

ALPHA	ABCDEFGHKL	PQRSTUVZ12
	Array Element 1	Array Element 2

The remainder of the internal record is blank-filled if the specified format does not satisfy the character count.

The following statement transfers a total of 32 characters from list elements A and B into an internal record whose starting location is denoted by ALPHA.

```
   ENCODE (32, 1, ALPHA) A,B
1  FORMAT (2(A10,A6))
```

Characters are transferred according to the specified format in statement 1 (ten characters from A(1), six from A(2), ten from B(1), and six from B(2)).

ALPHA ABCDEFGHIJ	KLMNOØPQRS	TUVWXYZ123	45
Array Element1	Array Element2	Array Element3	Array Element 4

DECODE is similar to READ and transfers the contents of an internal record into list elements using specified format conventions.

```
   DECODE (36, 3, GAMMA) A6, B6
3  FORMAT (2 (A10, A8))
```

The preceding DECODE statement transfers the contents of an internal record, whose starting location is denoted by GAMMA, into the list elements A6 and B6.

In the following example, the data contained within GAMMA are transferred according to the format specified in statement 3.

GAMMA	HEADERØ121	HEADØØ0131	HEADERØ122	HEADØØ0231
	Array Element1	Array Element2	Array Element3	Array Element 4

Therefore,

A6(1) will contain 'HEADERØ121'
A6(2) will contain 'HEADØØ01ØØ'
B6(1) will contain '31HEADERØ1'
B6(2) will contain '22HEADØØØØ'

Only eight characters of word 2 were transferred; however, since A and B were declared CHARACTER *10, the remainder is padded with blanks. The transfer continued for list element B6(1) with the remaining characters in the record.

11.13 BUFFERIN and BUFFEROUT (Asynchronous Input/Output)

BUFFERIN and BUFFEROUT are external subroutines used to access peripheral device buffers and perform asynchronous input/output operations. The specified number of words to be input or output is based on the type of record buffer element and the input/output device type.

Syntax

CALL BUFFERIN { u, m, a, w [, [n_1] [,n_2]] | u, m, a, w, [n_1] , [n_2] ,r }

CALL BUFFEROUT { u, m, a, w [, [n_1] [,n_2]] | u, m, a, w, [n_1] , [n_2] ,r }

u — The unit identifier, as specified in section 11.6.1.

m — An integer expression indicating the mode of operation (0=ASCII, 1=binary). In either mode, the data is unaltered. In binary mode, bits two and eight of the file control block word two are set. Refer to the MPX-32 Reference Manual for a description of the effects on various devices. Note, however, that there is no effect for a nine-track tape or disc device.

a — An array or variable to be used as the buffer for the operation. The transfer starts at the location specified and continues through w words contiguously. Note that if the buffer is too small, the following data or code locations will be destroyed. Normally this argument is an array name, but it can also be a variable or an array element.

w — An integer expression indicating the number of words to be input or output, starting at a.

n_1 The statement label of the first statement of an end-action routine to which control is returned when the software interrupt for input/output completion is honored. This routine must terminate with a call to the MPX-32 System Service X:XNWIO to exit the input/output end-action routine. Refer to the Scientific Run-Time Library Reference Manual for a description of FORTRAN-callable MPX-32 System Services; in particular, X:SYNCH and X:EAWAIT.

n_2 The statement label to which control is returned if hardware input/output errors are encountered. Refer to the Scientific Run-Time Library Reference Manual for a description of FORTRAN-callable MPX-32 System Services.

r A record specifier indicating the number of the record that is written to or read from the unit specified. This specifier is used exclusively for direct access input/output.

Rules for Use

- A BUFFERIN or BUFFEROUT operation always results in the processing of only one logical record of arbitrary length.
- If n_2 is specified and n_1 is not specified, a comma must be used to note the absence of n_1.
- If r is specified and n_1 or n_2 are not specified, two commas must be used to note the absence of n_1 and n_2.
- The bounding resulting from the type of the data buffer (INTEGER*4, INTEGER *2, etc.) must be compatible with the type of data transfer required by the device (disc/word transfer, tape/halfword transfer, etc.). (Example: For transfers to a disc device, the buffer should be of INTEGER*4 type, of REAL*4 type, or properly equivalenced to buffers of one of these types.)
- The actual number of words read by BUFFERIN may be less than the value of w.
- The maximum transfer value for E-class devices is based on the variable typing of the data buffer and the input/output device type. Table 11-3 correlates maximum transfer count, variable type, and input/output device type.

Table 11-3
Maximum Transfer Counts (E Class)

Variable Type (a)	I/O Device Type		
	Word	Halfword	Byte
Word	4095 words	2047 words	1023 words
Halfword	2047 words	2047 words	1023 words
Byte	1023 words	1023 words	1023 words

- The maximum transfer value for F-class devices are device dependent. The maximum value that can be specified in the w parameter is four megawords. However, the reference manuals for the particular I/O device and controller being used must be consulted for specific maximum transfer limitations.

- When using the r specifier, the unit specifier must be opened unblocked for direct access and must be assigned to a file.
- If more words are specified by w than exist in the record, the actual number of bytes transferred can be determined by a call to the external subroutine STATUS (refer to 11.14) or the function M:IOLEN (refer to the Scientific Run-Time Library Reference Manual). If the number of words specified by w is less than the number of words that exist in the record, the remaining words are ignored.
- When an end-of-file is encountered, magnetic tape will remain positioned immediately following the end-of-file. No data will be read into memory when an end-of-file is read. A subsequent CALL STATUS request will set the status to the value 3, and turn off the end-of-file indicator; a subsequent read or call to BUFFERIN will access the first record of the next file, if any.
- Data will have been read into memory by BUFFERIN if an input/output error occurred. A subsequent CALL STATUS will report a status indicator of 4.
- The physical buffer size n, written by BUFFEROUT, is set according to w, unless w exceeds the actual buffer size of the device being used. In that case, n is set according to the device buffer size.
- If a CALL STATUS indicates an error after a call to BUFFEROUT, the error will be an unrecoverable write error (the write operation will continue until completion).
- It is possible to execute any number of FORTRAN statements while the input/output transfers initiated by the BUFFERIN and BUFFEROUT statements are in progress. However, before attempting to use or modify the contents of the buffer, the user should perform either a CALL M:WAIT (refer to the Scientific Run-Time Library Reference Manual) or a CALL STATUS and wait for the status indicator to signal input/output completion. It is possible to execute both READ and WRITE statements during the BUFFERIN/BUFFEROUT operation, but only using a different unit identifier.
- Improper operation or status may result if consecutive BUFFERIN and BUFFEROUT statements are executed to the same unit identifier without an intervening CALL STATUS value of 2 being detected or a call to M:WAIT being made.
- If BUFFEROUT is used with the lfc assigned to Dev=UT, the 'enter CR for more' prompt will not appear when the page size is exceeded because BUFFEROUT is no-wait I/O.

Example

```
CALL BUFFERIN (50,0,I,80,100)
  :
100  FLAG=1
  :
CALL X:XNWIO
  :
```

11.14 CALL STATUS

CALL STATUS enables the user to test the status resulting from the latest input/output operation of any given unit.

Calling Sequence

CALL STATUS (lfc, status[,n])

lfc — An integer word variable that specifies the logical file code. The logical file code may be one to three ASCII characters, left-justified and blank-filled, or an integer constant.

status — An integer variable set according to the results of the status test as follows:

1. Not ready
2. Ready and no previous error
3. EOF or EOM sensed on latest input operation
4. Parity or lost data error on latest input/output operation
5. Unit is not open.

n — An integer variable that is set to the data transfer count. When the operation is complete, the data transfer count is in bytes. n is undefined if status= 5.

CHAPTER 12

FORMAT SPECIFICATION

12.1 General

A format used with formatted input/output statements provides information that directs the editing between the internal representation and the character strings of a record or sequence of records in the file. A format specification provides explicit editing information.

Formatting may also be influenced by the following:

- The NAMELIST statement, when referenced by a READ or WRITE statement, enables a user to input or output data values without specifying a format specification or argument list within the associated READ or WRITE statement.
- An asterisk (*) used as a format identifier in an input/output statement indicates list-directed formatting.

12.2 Format Specification Methods

Format specifications may be given:

- in FORMAT statements
- as values of character arrays, character variables, or other character expressions
- as character constants in input/output statements

12.2.1 FORMAT Statements

FORMAT statements are nonexecutable statements used with formatted input/output statements. They specify editing to be performed as data are transmitted between computer storage and external media.

Syntax

x FORMAT fs

x The mandatory statement label (one through five digits).

fs A format specification, as described in Section 12.3.

12.2.2 Format Specifications Stored in Variables and Arrays

Format specifications used with input/output statements may be given as values of character arrays, character variables, or other character expressions. If specifications are contained in a character variable, array, or array element, the appropriate name is used instead of a FORMAT statement label in the input/output statement.

The following example shows how the variables I and K and the array B are input according to format specifications read into the character array FMAT at execution time.

```
      CHARACTER*4 FMAT
      DIMENSION FMAT(5),B(5)
1     FORMAT (5A4)
      READ (3,1) FMAT
      READ (3,FMAT) I,K,(B(I),I=1,5)
        .
        .
        .
```

If the format identifier in a formatted input/output statement is a character array name, character variable name, or other character expression, the leftmost character positions of the specified entity must be in a defined state with character data that constitute a format specification when the statement is executed. In the preceding example, the first READ statement following statement label 1 insures that FMAT is in a defined state. The format information of input unit 3 might be the character string, (2I5/5F10.3), which indicates the I and K are to be read from one record and the array B from another.

If the format identifier is a character array name, the length of the format specification may exceed the length of the first element of the array; a character array format specification is considered to be a concatenation of all the elements of the array in the order given by array element ordering. However, if a character array element name is specified as a format identifier, the length of the format specification must not exceed the length of the array element.

12.2.3 Format Specifications Expressed as Character Constants

A format specification may be expressed as a character constant in a formatted input/output statement. Note that the form begins with a left parenthesis and ends with a right parenthesis. Character data may follow the right parenthesis that ends the format specification with no effect on the format specification. Blank characters may precede the format specification.

Example

```
WRITE (1,'(X,I6)')I
```

12.3 Form of a Format Specification

The form of a format specification is:

([flist])

flist A list of edit descriptors, as described in the following section.

12.3.1 Edit Descriptors

Edit descriptors determine the sizes of data fields and the type of presentation for each transmitted datum.

The format edit descriptors may have any of the following forms. However, the symbolic name of an integer cannot be used in a FORMAT statement that requires an integer constant.

Descriptor	Classification
srFw.d srEw.d srEw.dEe srGw.d srGw.dEe srDw.d rIw rIw.m rZw	Numeric edit descriptors
S SP SS	Output plus sign control
BN BZ	Input blank interpretation control
rA rAw	Character editing
'$h_1h_2...h_i$'	Apostrophe editing
n$Hh_1h_2...h_i$	Hollerith editing
rRw	R editing
rLw	Logical editing
nX Tn TLn TRn	Spacing specifications
:	Format control

w	is a positive integer constant defining the field width (including digits, decimal points, exponents, and algebraic signs) in the external data representation ($0 \leq w \leq 255$).
n	is a positive integer constant defining the number of characters or spaces ($0 \leq n \leq 255$).
d	is an integer specifying the number of digits appearing to the right of the decimal point (except for G editing) in the external data representation ($d \leq 255$ and $d \leq w$).
e	is a nonzero, unsigned integer constant specifying the exponent part of E and G edit descriptors.
F,E,G,D, I,Z,A,R,L	indicate the type of editing to be applied to the items in an I/O list.
H,X,T, TL,TR,:, S,SP,SS, BN,BZ,/	specify information that is provided directly from the format descriptor The specifications containing these characters have no corresponding I/O list items.
r	is an optional, positive integer constant, indicating the number of times the descriptor will be repeated ($0 \leq r \leq 255$).
s	is an optional scale factor of the form nP where n is a signed integer constant ($-77 \leq n \leq 76$). Note that the scale factor s of the form nP does not have to be attached to a numeric edit descriptor.
h_i	are characters from the ASCII character set.
m	specifies the minimum number of output digits to be generated.

12.3.2 Interpretation of Blanks on Input

The interpretation of blanks on input is influenced by FORTRAN 77+ run-time option 8, which may be set at catalog time or upon execution. Option 8 provides compatibility with FORTRAN-IV in the following way. The interpretation of embedded blanks in an input field defaults to zero (0) if option 8 is in effect when an implicit open is executed; if this option is not in effect, the interpretation of blanks defaults to NULL. To override the default, use a BLANK= specifier in an OPEN statement, or a BN or BZ edit descriptor in a FORMAT statement.

12.4 Format Control List Specifications and Record Demarcation

The following relationships and interactions between format control, input/output lists, and record demarcation should be noted:

- Execution of a formatted READ or WRITE statement initiates format control.
- The editing performed on data depends on information jointly provided by the elements in the input/output list and field descriptors in the FORMAT statement.
- If there is an input/output list, at least one descriptor of types D, E, F, G, I, L, Z, R, or A must be present in the FORMAT statement.
- Each execution of a formatted READ statement causes one or more new records to be input.
- Each item in an input list corresponds to a string of characters in the record and to a descriptor of the types D, E, F, G, I, L, Z, R, or A in the FORMAT statement.
- H editing, apostrophe editing, Hollerith editing, and T and X editing communicate information directly between the external record and the field descriptors without reference to list items.
- On input, when a slash is encountered in the FORMAT statement, or when the format descriptors have been exhausted and reuse of the descriptors is initiated, processing of the current record is terminated and the following occur:
  - Any unprocessed characters in the record are ignored.
  - If more input is necessary to satisfy list requirements, the next record is accessed.
- A READ statement is terminated when all items in the input list have been satisfied if:
  - The next format descriptor is D, E, F, G, I, L, Z, R, A, or a colon (:) descriptor.
  - The format control has reached the last outer right parenthesis of the FORMAT statement.

  If the input list has been satisfied, and the next FORMAT descriptor is H, T, TL, TR, /, or X, that descriptor is processed (with the possibility of new records being entered) until one of the above conditions exists.
- If the number of list items is less than the number of format descriptors, the processor matches from left to right a list item with a format descriptor; any remaining format descriptors are ignored.

- When a formatted WRITE statement is executed, records are written each time a slash is encountered in the FORMAT statement or format control has reached the rightmost parenthesis. The format control terminates in one of the two methods described for READ termination. For unblocked or blocked files, records may be padded or truncated, as required by a device.

```
      READ (5, 1, END=99) L, M, N
    1 FORMAT (I3, I1)
```

  These statements obtain the value of L from positions 1-3 of record 1, the value of M from position 4 of record 1, and the value of N from positions 1-3 of record 2.

- For formatted READ statements, a comma or the end of the record, indicated by the ASCII character with the hexadecimal representation X'0D' (carriage return), may be used to terminate any numeric field. Thus, the data read need not be the full length described in the FORMAT statement.

12.5 Forms Control on Output

When the unit to which formatted output is directed is an online terminal or line printer, the first character of each formatted record is used for forms control. In this mode, the first character of each record is not printed. Instead, it controls the printer vertical spacing.

Control Character	Terminal	Line Printer
Blank	Linefeed/carriage return before write.	Single space before print.
0	Two linefeeds/carriage returns before write.	Doublespace before print.
1	Five linefeeds/carriage returns before write.	Eject page before print.
+	No linefeed before write; No carriage return before write.	No space before print.
<	Linefeed/carriage return before write.	Single space before print. Inhibit header output for the SLO file.
>	Linefeed/carriage return before write.	Single space before print. Enable header output for the SLO file.
-	Five linefeeds/carriage returns before write.	Eject page and print user specified title record, beginning with the second character*.
=	Linefeed/carriage return before write.	Eject page before print. Clear user specified title output for SLO file.

* For spooled printer output, a minus sign as the first character of a record causes a title buffer to be filled from the record; a maximum of three title line buffers may be filled from a maximum of three successive lines, beginning with a minus sign.

All other characters appearing as the first character of the record will be treated as if they were blanks and will cause vertical spacing of one line.

Care must be taken in preparing line printer output; the first character of the standard length buffer is used solely as the vertical control character and will not be printed. The statements

```
10 FORMAT (1X, 'THIS␣IS␣PROGRAM␣A')
```

and

```
15 FORMAT ('␣THIS␣IS␣PROGRAM␣A')
```

are equivalent in their resulting output. There will be vertical spacing of one line (i.e., the normal spacing of skipping to the next line), after which the following is printed:

```
THIS IS PROGRAM A
```

12.6 Numeric Editing

Numeric edit descriptors are used to specify the input/output of integer, real, double precision, and complex data as well as the interpretation of blanks (other than leading blanks) in numeric input fields.

12.6.1 D Editing and Output

Form: Dw.d

Double precision and complex doubleword data are processed with this edit descriptor. w characters are processed, of which d are considered fractional.

The external output format is the same as E output format except the letter E is replaced by the letter D.

Examples

Format Descriptor	Internal Value	Output (␣=blank)
D16.9	+12.34567890	␣␣.123456789D+02
D15.8	-0.0012763	␣-.12763000D-02
D15.4	-9.176	␣␣␣␣-.9176D+01
D12.6	+123567	␣.123567D+06

12.6.2 D Input

Data values that are to be processed under D, E, F, or G edit descriptors can be in a relatively loose format in the external input medium. For example, each of the following is optional:

- Leading spaces (ignored)
- + sign (an unsigned input is assumed positive)
- A string of digits
- A decimal point
- A second string of digits
- The character D or E
- A signed decimal exponent

Input data can be any number of digits in length, but precision will be maintained only to the level specified in Chapter 3 for real or double precision data.

Examples

When no decimal point is given among the input characters, the d in the format specification establishes the location of the implied decimal point in the input item. If a decimal point is included in the input characters, the d specification is ignored in determining the location of the decimal point.

The letters D, E, F, and G are interchangeable in format specifications or in the input data. The end result is the same. All input values under these specifications are formatted first to double precision format and then truncated to fit the defined data typing, if required. Thus, D, E, F, or G may enter either double or single precision values.

Format Descriptor	Input (b=blank)	Internal Value
D14.3	bbbbbbbbbbbbbbb	+0.000
D10.3	bbbbbbbbb5.	+5.000
D10.3	bbbb3bbbbb	+300.0 (if BZ is in effect) .003 (if BN is in effect)
D10.3	bb1.276Eb4	+12760
D10.3	-763267E-3	-.763267
D10.3	bb1.276D2b	+1.276D20 (if BZ is in effect) +1.276D2 (if BN is in effect)
E10.3	+0.23756+4	+2375.60
E10.3	bbbbb17631	+17.631
F8.3	b1628911	+1628.911
F12.4	bbbb-6321132	-632.1132
G10.1	bbbb167223	+16722.3
G10.0	bbbbb-1263	-1263.0

12.6.3 E Editing

Form: Ew.d

Real and complex type data are processed using this edit descriptor. w characters are processed, of which d are considered fractional.

Form: Ew.dEe

Real and complex type data are processed using this edit descriptor. w characters are processed, of which d are considered fractional. The exponent part consists of e decimal digits. The e has no effect on input.

12.6.4 E Output and Input

Values are formatted, rounded to d digits, and output in order, as

- A minus sign if negative (plus sign if positive and if SP descriptor is in effect).
- A decimal point.
- d decimal digits.
- The letter E.
- The sign of the exponent.
- e exponent digits; in the first form, e defaults to two (note that the absolute value of the exponent is always ≤ 79).

The values, as described, are right-justified in the field w, with preceding blanks to fill the field if necessary. The field width w must satisfy the following relationship; otherwise the output field is filled with asterisks.

$w \geq$ d+e+3 if the SP descriptor is not in effect and the value is not negative.
d+e+4 if the SP descriptor is in effect or the value is negative.

For E input, refer to the description of D input.

Examples

Format Descriptor	Internal Value	Output (ϕ=blank)
E12.5	76.573	ϕϕ.76573E+02
E13.7	-32672.354	-.3267236E+05
E7.3	56.93	*******
E13.4	-0.0012321	ϕϕϕ-.1232E-02
E8.2	-3.567	-.36E+01
E8.2	76321.73	ϕ.76E+05
E12.5E02	76.573	ϕϕϕ.76573E+2

12.6.5 F Editing

Form: Fw.d

Real and complex type data are processed using this edit descriptor. w characters are processed, of which d are considered fractional.

12.6.6 F Output and Input

Values are formatted and output as a minus sign (if negative), a plus sign (if positive and if SP descriptor is in effect) followed by the integer portion of the number, a decimal point, and d digits of the fractional portion of the number. If a value does not fill the field, it is right-justified and preceding blanks to fill the field are inserted. If a value requires more field positions than allowed by w, the output field is filled with asterisks. Therefore, the field width w should satisfy the relationship

$$w \geq d + 3$$

For F input, refer to the description of D input.

Examples

Format Descriptor	Internal Value	Output (␣=blank)
F10.4	368.42	␣␣368.4200
F7.1	-4785.361	-4785.4
F8.4	3.75E-2	␣␣0.0375
F6.4	4739.76	******
F7.3	-5.6	␣-5.600

12.6.7 G Editing

Form: Gw.d

Real and complex data are processed using this edit descriptor. The descriptor indicates that the external field occupies w character positions with d significant digits.

Form: Gw.dEe

Real and complex data are processed using this edit descriptor. The descriptor indicates that the external field occupies w character positions with d significant digits. The exponent part consists of e decimal digits. The e has no effect on input.

12.6.8 G Output and Input

The form of output depends on the magnitude of the number being edited. The following table shows the editing used.

Number Magnitude	Equivalent Editing Effected
$0.1 \leq N < 1$	F(w-n).d,nX
$1 \leq N < 10$	F(w-n).(d-1),nX
.	.
.	.
.	.
$10^{d-2} \leq N < 10^{d-1}$	F(w-n).1,nX
$10^{d-1} \leq N < 10^{d}$	F(w-n).0,nX
Otherwise	sEw.d or sEw.dEe

n is 4 for Gw.d and e+2 for Gw.dEe. (Refer to 12.10 for an explanation of sEw.d and sEw.dEe.)

For G input, refer to the description of D input.

Examples

Format Descriptor	Internal Value	Output ␣=blank
G12.4	+.056321	␣␣␣.5632E-01
G12.4	-.563217	␣␣-.5632␣␣␣␣
G12.4	+5.63217	␣␣␣5.632␣␣␣␣
G12.4	+56321.7	␣␣␣.5632E+05
G10.4	+563.217	␣563.2␣␣␣␣

12.6.9 Complex Editing

Complex numbers are made up of two single precision real numbers. Each of the numbers is described using D, E, F, or G format descriptors. For each complex item in the input/output list, there must be a pair of format descriptors in the associated FORMAT statement.

Example

```
COMPLEX COMP1, COMP2
 .
 .
 .
READ (3,20) I, COMP1, COMP2
 .
 .
 .
20  FORMAT (I6,   F8.2,F7.3,      F8.2,F7.3)
                  ASSOCIATED      ASSOCIATED
                  WITH COMP1      WITH COMP2
```

12.6.10 I Editing

Form: Iw

Only integer data may be formatted by this form of edit descriptor. w specifies field width.

Form: Iw.m

Only integer data may be formatted by this form of edit descriptor. w specifies field width. m specifies that the unsigned integer constant w must consist of at least m digits including, if necessary, leading zeros. The value of m must not exceed the value of w. The m has no effect on input. If the value of m is zero and the value of the internal datum is zero, the output field consists only of blank characters, regardless of the sign control in effect.

12.6.11 I Output

Values are formatted to integer constants. Negative values are preceded by a minus sign (or a plus sign if SP is in effect). If the value does not fill the field, it is right-justified and enough preceding blanks to fill the field are inserted. If the value exceeds the field width, the field is filled with asterisks (*).

Integer variables may be byte, halfword, fullword, and doubleword in length. The I format descriptor will be equally valid for all of these variable types.

Format Descriptor	Internal Value	Output (ϕ=blank)
I6	+281	ϕϕϕ281
I6	-43261	-43261
I3	126	126
I3	-226	***
I3	46931	***
I6.4	+281	ϕϕ0281

12.6.12 I Input

A field w characters long is entered and formatted to internal integer format. A minus sign may precede the integer digits. If a sign is not present, the value is considered positive.

Doubleword integer values in the range -9223372036854775808 to +9223372036854775807 are accepted. The input item is treated as an integer number with an implicit decimal point immediately to the right of the field of w characters.

Format Descriptor	Input (ϕ=blank)	Internal Value
I4	ϕ124	124
I4	-124	-124
I7	ϕϕ6732ϕ	67320 (if BZ is in effect) 6732 (if BN is in effect)
I4	1ϕ2ϕ	1020 (if BZ is in effect) 12 (if BN is in effect)

12.6.13 Z Editing

Form: Zw

This descriptor processes all types of data. It causes hexadecimal values to be read into or from a specified list item.

12.6.14 Z Output

The maximum hexadecimal digits that may be transmitted between internal and external representations using Zw is two times the number of storage units required by the corresponding list item (for example, 8 digits for real items and 16 digits for double precision).

Z Input/S, SP, and SS Descriptors

Format Descriptor	Decimal Value	Type	Output Value
Z4	516	INTEGER	0204
Z4	-2	INTEGER	FFFE
Z8	1.5	REAL	41180000
Z8	-1.5	REAL	BEE80000
Z9	-1.5	REAL	ØBEE80000
Z7	-1.5	REAL	EE80000

If the specified field size is less than the number of hexadecimal digits that the list item holds, the leftmost hexadecimal digit(s) will be lost. The last line above is an example of this data loss.

12.6.15 Z Input

Since one storage location (byte) in internal storage contains two hexadecimal digits, if an input field contains an odd number of digits, the number will be padded on the left with a hexadecimal zero when it is stored. If the number to be entered is larger than the storage location, only the rightmost digits will be input.

Format Descriptor	Input Value	Type	Decimal Value
Z4	0204	INTEGER	516
Z9	ØBEE80000	REAL	-1.5
Z2	FF	INTEGER	255

12.6.16 S, SP, and SS Descriptors

The S, SP, and SS descriptors may be used to control optional plus characters in numeric output fields. At the beginning of execution of each formatted output statement, the processor normally produces a blank rather than a plus in numeric output fields.

If an SP descriptor is encountered in a format specification, the processor must produce a plus in any subsequent position that optionally may contain a plus sign.

If an SS descriptor is encountered, the processor must not produce a plus in any subsequent position that optionally may contain a plus sign.

If an S descriptor is encountered, the option of producing a plus is restored to the processor, thus, the sign is noted by a blank rather than a plus.

The S, SP, and SS descriptors affect only I, F, E, D, and G editing during the execution of an output statement. These three descriptors have no effect during the execution of an input statement.

For example,

Format Descriptors	Internal Value	Output (ß=blank)
SP,G12.4	+.056321	ß+0.5632E-01
SS,F10.4	368.42	ßß368.4200
S,I6	+281	ßßß281

In the first line, the SP descriptor appears, so the output includes the plus sign. In the second line, the SS descriptor causes the optional plus to be suppressed. In the third line, the S descriptor causes the optional + to be suppressed.

12.6.17 BN and BZ Descriptors

The BN and BZ descriptors may be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of each formatted input statement, such blank characters are interpreted as zeros or are ignored, depending on the value of the BLANK specifier in effect for the unit. Note that the interpretation of blanks defaults to zero if run-time option 8 is in effect on an implicit open.

If a BN descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields for this FORMAT statement are ignored; the effect of a BN descriptor does not carry over to subsequent READ or ACCEPT statements. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right-justified, and the blanks replaced as leading blanks. However, a field of all blanks has the value zero.

If a BZ descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields for this FORMAT statement are treated as zeros; the effect of a BZ descriptor does not carry over to subsequent READ or ACCEPT statements.

The BN and BZ descriptors affect only I, F, E, D, G, and Z editing during the execution of an input statement. They have no effect during the execution of an output statement.

For example,

Format Descriptors	Input (ß=blank)	Internal Value
BN,I5	ßß7ßß	+7
BN,D10.3	ßßßß3ßßßßß	+0.003
BZ,I5	ßß7ßß	+700
BZ,D10.3	ßßßß3ßßßßß	+300.0

12.7 Character Editing

Character edit descriptors are used to specify the input/output of character data.

12.7.1 A Editing

Form: Aw

This descriptor causes unmodified bytes to be read into or written from a specified list item; each byte can be typically considered as representing an ASCII character.

The maximum number of actual characters that may be transmitted between internal and external representations using Aw is 255.

Form: A

This descriptor causes unmodified ASCII characters to be read into or written from a specified list item.

If a field width w is not specified with the A edit descriptor, the number of characters in the field is the length of the character I/O list item.

12.7.2 A Output

If w is greater than n (where n is the number of storage units required by the list item), the ASCII string will be right-justified and blank-filled in a field of width w. If w is less than n, the external output field will consist of the leftmost w characters from the internal representation.

Format Descriptor	Internal (ϕ=blank)	Type	Output (ϕ=blank)
A	ABCD	CHARACTER	ABCD
A	ABCD1234	CHARACTER	ABCD1234
A4	ABCDE	CHARACTER	ABCD
A5	AB	CHARACTER	ϕϕϕAB

12.7.3 A Input

If w is greater than n (where n is the number of storage units required by the corresponding list item), the rightmost n characters are taken from the external input field. If w is less than n, the w characters appear left-justified, with w-n trailing blanks in the internal representation.

Format Descriptor	Input Characters	Type	Internal (␣=blank)
A	ABCD	CHARACTER	ABCD
A4	ABC	CHARACTER	ABC␣
A3	ABCD	CHARACTER	ABC

12.7.4 Apostrophe Editing

Form: '$h_1h_2...h_n$'

The characters in the field bounded by apostrophes (') are placed in the external field. Any number of characters may be in a character string. Blanks are counted as characters and will be included in the output character stream. The use of a quotation mark as an escape character in the literal format descriptor is discussed in Chapter 3.

Format Descriptor	Output (␣=blank)
'␣X␣'	␣X␣
'X'	X
'SHOULDN''T'	SHOULDN'T

12.7.5 H Editing

Form: $nHh_1, h_2,...,h_n$

12.7.6 H Output

The n characters are placed in the external field. The number of characters in the string must be exactly as specified by n. Otherwise, characters from other descriptors may be taken as part of the string. For example, if the descriptor 4H1,I2 appears, an error will not be detected. However, if 2H123 is used, an error will be detected in the compiler and a message will appear in the diagnostics. Blanks are counted as characters.

Format Descriptor	Output (␣=blank)
1HR	R
8H␣STRING␣	␣STRING␣
12HX␣(1,3)=12.0	X␣(1,3)=12.0
11HI␣SHOULDN'T	I␣SHOULDN'T

12.7.7 A Editing with Noncharacter Data Types

The A edit descriptor should be used with an input/output list item of type CHARACTER to comply with the ANSI X3.9 - 1978 standard; however, FORTRAN 77+ allows the use of this edit descriptor with noncharacter data types.

Form: Aw

The maximum number of actual characters that may be transmitted between internal and external representations using Aw is 255.

Form: A

If a field width w is not specified with the A edit descriptor, the number of characters in the field is the length of the character I/O list item.

12.7.8 A Output with Noncharacter Data Types

If w is greater than n (where n is the number of storage units required by the list item), the ASCII string will be right-justified and blank-filled in a field of width w. If w is less than n, the external output field will consist of the leftmost w characters from the internal representation.

Format Descriptor	Type	Internal Storage	Output
A2	INTEGER	X'24312020'	$1
A3	INTEGER	X'41424320'	ABC
A4	REAL	X'41424344'	ABCD
A5	REAL	X'41424344'	␣ABCD
A3	REAL	X'41424344'	ABC
A8	COMPLEX	X'4142434431323334'	ABCD1234
A6	DOUBLE PRECISION	X'4142434431323334'	ABCD12

12.7.9 A Input with Noncharacter Data Types

If w is greater than n (where n is the number of storage units required by the corresponding list item), the rightmost n characters are taken from the external input field. If w is less than n, the w characters appear left-justified, with w-n trailing blanks in the internal representation.

Format Descriptor	Type	Input Characters	Internal Storage
A1	INTEGER*2	$	X'2420'
A3	INTEGER	ABC	X'41424320'
A1	REAL	A	X'41202020'
A5	REAL	ABCDE	X'42434445'
A6	COMPLEX	ABCDEF	X'4142434445462020'

12.7.10 R Editing

Form: Rw

This descriptor causes unmodified bytes to be read into or written from a specified list item; each byte can be typically considered as representing an ASCII character.

The maximum number of actual characters that may be transmitted between internal and external representations using Rw is the same as the number of storage units required by the corresponding list item (for example, four characters for real items, eight characters for double precision).

12.7.11 R Output

If w is greater than n (where n is the number of storage units required by the list item), the ASCII string will be left-justified and blank-filled on the right in a field of width w. If w is less than n, the external output field will consist of the rightmost w characters from the internal representation.

Format Descriptor	Type	Internal	Output (ϕ=blank)
R2	INTEGER (INTEGER*4)	X'20202431'	$1
R3	INTEGER (INTEGER*4)	X'20414243'	ABC
R4	REAL (REAL*4)	X'41424344'	ABCD
R5	REAL (REAL*4)	X'41424344'	ABCDϕ
R3	REAL (REAL*4)	X'41424344'	BCD
R8	DOUBLE PRECISION (REAL*8)	X'4142434431323334'	ABCD1234
R5	DOUBLE PRECISION (REAL*8)	X'4142434431323334'	D1234

12.7.12 R Input

If w is greater than n (where n is the number of storage units required by the corresponding list item), the leftmost n characters are taken from the external input field. If w is less then n, the w characters appear right-justified with leading binary zeros in the internal representation.

Format Descriptor	Type	Input Characters	Internal
R1	INTEGER*2	$	X'0024'
R3	INTEGER (INTEGER*4)	ABC	X'00414243'
R1	REAL (REAL*4)	A	X'00000041'
R5	REAL (REAL*4)	ABCDE	X'41424344'
R6	DOUBLE PRECISION (REAL*8)	ABCDEF	X'0000414243444546'

12.8 Logical Editing

Form: Lw

The Lw edit descriptor indicates that the field occupies w positions. The specified input/output list item must be of type logical. On input, the list item will become defined with a logical. On output, the specified list item must be defined with a logical.

12.8.1 L Output

The output field consists of w-1 blanks followed by a T or F as the value of the internal datum is .TRUE. or .FALSE., respectively.

Format Descriptor	Internal Value	Output (ø=blank)
L1	.FALSE.	F
L1	.TRUE.	T
L5	.TRUE.	øøøøT
L7	.FALSE.	øøøøøøF

12.8.2 L Input

A string of w characters is entered. Leading blanks and an optional period as the first nonblank character are ignored. If the first accepted character of the input string is T, the value .TRUE. is assigned to the input item. If the first accepted character is other than T, or if no characters are present, the value .FALSE. is assigned to the list item. All other characters in the field are ignored.

Format Descriptor	Input (ø=blank)	Resultant Values Logical
L1	T	.TRUE.
L1	.	.FALSE.
L2	.T	.TRUE.
L4	ø.FA	.FALSE.
L6	øøøø.T	.TRUE.
L8	øFALSEøø	.FALSE.
L6	øTESTø	.TRUE.
L3	.øT	.FALSE.

12.9 Positional Editing

The positional edit descriptors specify where the next character will be transmitted to or from the record.

12.9.1 The X Descriptor

Form: nX

The X descriptor causes no editing to occur, nor does it correspond to an item in an input/output list. When used for output, it causes n blanks to be inserted in the output record. When used with input, this descriptor causes the next n characters of the input record to be skipped. Note that n is optional; if n is not present, the default is 1.

Output Examples

Format Statement	Output List Value	String (Ø=blank)
3 FORMAT('X=',2X,I3)	-27	X=ØØ-27
5 FORMAT(3X,'READ',5X,'WRITE')		ØØØREADØØØØØWRITE

Input Examples

Format Statement	Input String	Resultant Input
10 FORMAT(F4.1,3X,F4.1)	12.5ABC120.	12.5,120.0
15 FORMAT(7X,I3)	12.5ABC120	120

12.9.2 The T Descriptor

Form: Tn

The T descriptor causes no editing to occur, nor does it correspond to an item in an input/output list; it is a means of tabulating input/output streams. n is an integer constant between 1 and 255 specifying the character position in the record where the next data transfer begins.

Form: TLn

The TL descriptor indicates that the transmission of the next character to or from the record is to occur at the character position n characters backward (left) from the current position, not to precede the leftmost character position.

Form: TRn

The TR descriptor indicates that the transmission of the next character to or from the record is to occur at the character position n characters forward (right) from the current position.

Output Examples

Format Statement	Result
15 FORMAT (I4,2X,G12.2,T76,I5)	One integer is output, two spaces are skipped, one real number is output, then a skip is made to the 76th position, where one integer is output. Blanks will appear in the parts of the record that were skipped.
15 FORMAT (I4,2X,G12.2,TR3,I5)	One integer is output, two spaces are skipped, one real number is output, then a skip is made three spaces to the right of the current position, where one integer is output. Blanks will appear in the parts of the record that were skipped.

Input Examples

Format Statement	Result
10 FORMAT (G8.0,T1,I8,T1,Z8)	Allows the input of the same record item(s) through multiple format descriptors, since each T1 causes a return to the beginning of the input record.

12.10 Scale Factors

To provide more flexible use of D, E, F, and G edit descriptors, a scale factor designator may precede these format specifications.

Form: sDw.d sEw.d sFw.d sGw.d

s is the form nP, with n an integer constant with a range of $-77 \leq n \leq +76$

A scale factor of zero is established when format control is initiated for each READ or WRITE statement. Once established, a scale factor applies to all subsequent D, E, F, and G editing in this format until a new factor is defined.

Example

```
7 FORMAT (3PE7.3,F4.1,2PD7.2,E7.1)
```

In this example, the scale factor 3P is in effect for the first two descriptors and changed to 2P by the third. 2P is still effective for the fourth descriptor.

In the following examples and discussions use the parameter definitions described below.

w The field width.

d The number of digits in the fractional part of the external field.

n The integer constant portion of the scale factor nP.

The scale factor affects editing in the following manner:

- On input, if the datum (D, E, F, or G edit descriptor) has an explicit exponent, the scale factor has no effect. If the datum has no explicit exponent, the scale factor editing formula is

  input datum * 10^{-n} =internal value

  n is the scale factor integer constant.

Format Descriptor	Input Datum	Internal Value
2PE12.4	ØØ0.7632E-03	.0007632
3PE12.3	ØØØØ+7.732D3	7732.
2PE12.4	ØØØØØØ763621	.763621
1PF8.3	-123.762	-12.3762
-2PG10.3	246.731	24673.1

- On output, using E or D edit descriptor, the basic real constant part of the quantity to be produced is multiplied by 10^n and the exponent is reduced by n. In addition, the scale factor n controls decimal normalization as follows:

  If $-d < n \leq 0$, the output field contains exactly |n| leading zeros and d- |n| significant digits after the decimal point. For example, the format descriptor

  ```
  -1PE11.2
  ```

  causes the output field to contain one leading zero and one significant digit to the right of the decimal point.

  If $0 < n < d+2$, the output field contains exactly n significant digits to the left of the decimal point and d-n+1 significant digits to the right of the decimal point. For example, the format descriptor

  ```
  3PE11.2
  ```

  causes the output field to contain three significant digits to the left of the decimal point and 0 significant digits to the right of the decimal point.

Other values of n for output are not permitted.

Format Descriptor	Stored Value	External Representation
-1PE11.2	12.7	ØØØ01.3E01
3PE11.2	12.7	ØØØ127.E-01
4PD12.3	12.7	ØØØ1270.D-02

- On output using F editing, the stated value is multiplied by 10^n, actually altering the external value.

Format Descriptor	Stored Value	Internal Representation
3PF11.2	12.7	ØØØ12700.00
-2PF11.5	-.05634	ØØØ-.00056

- On output using G editing, the method of representing the output field depends on the magnitude of the datum being edited as follows:

  If the magnitude of the datum is less than 0.1, or greater than or equal to 10**d, G output editing is the same as E output editing, i.e., sPGw.d is equivalent to sPEw.d and sPGw.dEe is equivalent to sPE.dEe, where s is the scale factor currently in effect.

Format Descriptor	Stored Value	External Representation
2PG12.4	.056321	ØØ56.321E-03
2PG12.4E3	.056321	Ø56.321E-003

- The scale factor has no effect on I, A, R, L, H or Z editing.

12.11 Repeat Specifications

The D, E, F, G, I, L, Z, R, and A field descriptors may be indicated as repetitive descriptors by using a repeat count r in the form rDw.d, rEw.d, rFw.d, rGw.d, rIw, rLw, rZw, rRw, and rAw.

The following pairs of FORMAT statements are equivalent:

```
 2 FORMAT (3F8.3,F9.2)

 2 FORMAT (F8.3,F8.3,F8.3,F9.2)

18 FORMAT (2I6,3A7,2E12.5)

18 FORMAT (I6,I6,A7,A7,A7,E12.5,E12.5)
```

Repetition of a group of field descriptors is accomplished by enclosing the group in parentheses, preceded by a repeat count. The absence of a repeat count indicates a count of one. Five levels of parentheses, not counting the parentheses required by the FORMAT statement, are permitted. A repeat count may not exceed 255.

The following statements are equivalent for transmitting up to four variables:

```
3 FORMAT (I7,3(F8.1,4X))

3 FORMAT (I7,F8.1,4X,F8.1,4X,F8.1,4X)
```

Repetition of format descriptors is also initiated when all descriptors in the FORMAT statement have been used but there are still items in the input/output list that have not been processed; the format descriptors are reused, starting at the opening parenthesis that immediately precedes the last closing parenthesis in the FORMAT statement. The parentheses enclosing the entire list of descriptors are not considered, unless there are no other parentheses in the list. A repeat count preceding the parenthesized descriptor(s) is reused if the descriptors are to be reused. When the reuse of a parenthesized descriptor is initiated, processing of the current record terminates, and a new record is processed.

Input Example

```
      DIMENSION B(100)
      READ (3,75) B
           .
           .
           .
75  FORMAT (5F9.2)
```

In the previous example, the first five fields are taken from each of 20 records (so as to total 100, the number of array elements) and are input and assigned to the array elements of the array B.

Output Example

```
           .
           .
           .
      WRITE (7,9) E,F,K,L,M,KK,LL,MM,K3,L3,M3
9  FORMAT (2F9.3,(3I7))
```

In this example, three records are written.

Record 1 contains the values of E, F, K, L, M, record 2 contains the values of KK, LL, MM, and record 3 contains the values of K3, L3, M3 because the descriptor 3I7 is used three times.

12.12 Field Separators

Two adjacent descriptors must normally be separated in the FORMAT statement by either a comma or one or more slashes. Commas may be omitted in the following cases:

- Following Hollerith or blank field descriptors
- Between an nP descriptor and a following D,E,F,G descriptor
- Before or after a colon (:) descriptor

The slash not only separates field descriptors, but it also specifies the demarcation of formatted records.

```
1X,2H0K,F6.3
     and              are equivalent and specify a single record
1X2H0KF6.3

1X,2H0K/F6.3          specifies two records
```

Each slash terminates the current record and initiates processing of the next record. The remaining input record is ignored; the remaining output record is filled with blanks. Successive slashes (///.../) cause successive records to be ignored on input and successive blank records to be written on output.

Output Example

```
      DIMENSION A(20), J(20)
           :
           :
      WRITE (7,8) J,A
    8 FORMAT (10I7/10I7/10F7.3/
     X                10F7.3)
```

In this example, the data specified by the list of the WRITE statement are output to the specifications of FORMAT statement 8. Four records are written as follows:

Record 1:	J(1)	J(2)	J(3)	...J(10)
Record 2:	J(11)	J(12)	J(13)	...J(20)
Record 3:	A(1)	A(2)	A(3)	...A(10)
Record 4:	A(11)	A(12)	A(13)	...A(20)

Input Example

```
      DIMENSION B(10)
           :
           :
      READ (4,17) B
           :
           :
   17 FORMAT (F10.2/F10.2///8F10.2)
```

In this example, the two elements B(1) and B(2) of array B receive their values from the first data fields of successive records, and the remaining elements of the two records are ignored. The third and fourth records are ignored, and the remaining elements of the array are filled from the fifth record.

12.13 Colon Descriptor

The colon descriptor (:) terminates format control if no items remain in the I/O list. The colon descriptor has no effect if there are more items in the I/O list.

Example

```
      N = 3
      K(1)=34
      K(2)=4
      K(3)=42
      WRITE ('LO',100) (K(I), I=1,N)
100   FORMAT (50(' INVENTORY IS',I6,' UNITS':/))
      WRITE ('LO', 101)
101   FORMAT (' END OF LIST')
```

The preceding sequence of statements would produce the following sample printout:

```
INVENTORY IS          34 UNITS
INVENTORY IS           4 UNITS
INVENTORY IS          42 UNITS
END OF LIST
```

In the preceding example, if the colon descriptor were not used the printout would appear as follows:

```
INVENTORY IS          34 UNITS
INVENTORY IS           4 UNITS
INVENTORY IS          42 UNITS
INVENTORY IS
END OF LIST
```

CHAPTER 13

EXTENDED ADDRESSING

13.1 Introduction

Three FORTRAN 77+ specification statements support extending addressing:

- EXTENDED BLOCK
- EXTENDED BASE
- EXTENDED DUMMY

The statements enable the use of configured main memory between 512 KB and 16 MB for storing/retrieving data items that are bound to absolute logical address locations at compile time. For larger commons, extended addressing must be used. The GLOBAL COMMON may be as large as extended memory, i.e., .5MB on a 32/77, 1.5MB on a 32/27 or 32/87, and 15.5MB (minus the operating system) on a 32/67 or 32/97.

13.2 EXTENDED BLOCK Statement

A data area can be established in the user's extended data space by using the EXTENDED BLOCK statement.

Syntax

EXTENDED BLOCK [/y_1/] a_1 [,/y_2/a_2,. . .,/y_i/a_i]

y_i EXTENDED BLOCK storage names, which can be any FORTRAN 77+ symbolic name other than GLOBAL00 through GLOBAL99.

a_i Sequences of variable names, array names, or constant array declarators, separated by commas.

The elements in a_i make up the EXTENDED BLOCK storage area specified by the name y_i.

If any y_i is omitted (two consecutive slashes //), the block of storage so indicated is called BLANK EXTENDED. If the first block name (y_1) is omitted, the two slashes can be omitted. BLANK EXTENDED always begins at the first page of extended memory. It is referred to by empty block name specifications. For the definition of any specific extended block, an extended block name can appear more than once in the same EXTENDED BLOCK statement or in more than one EXTENDED BLOCK statement within a program unit. EXTENDED DUMMY statements are used in subprograms instead of EXTENDED BLOCK statements to reference the extended arguments defined in the program unit. Extended common areas in subroutines are still defined using the EXTENDED BLOCK statement.

Variables can be assigned to EXTENDED BLOCK storage in the same manner as for a COMMON block or by using an EQUIVALENCE statement.

A GLOBAL memory partition (either static or dynamic) may be used as the EXTENDED BLOCK name. However, the name of the partition must be other than GLOBAL00 through GLOBAL99; for example, EXTEND01.

Access to an EXTENDED BLOCK can be gained by use of the X:GDSPCE service to obtain map blocks spanning the desired address range. Alternatively, an extended memory partition can be created by using the volume manager CREATE COMMON directive; access is then gained by means of the X_INCLD service.

Items that are assigned to an EXTENDED BLOCK cannot also appear in a separate COMMON statement.

13.3 EXTENDED BASE Statement

The EXTENDED BASE statement allocates an extended block to a specific address range in the user's extended data space.

Syntax - 1

EXTENDED BASE/y_1/p_1 [,] /y_2/p_2. . .[,] /y_i/p_i

y_i Symbolic names other than GLOBAL00 through GLOBAL99, or DATAPOOL.

p_i Positive integer values of 256 or greater that specify a logical memory page.

An EXTENDED BASE statement referencing a common block name allocates the corresponding common block into extended memory.

In the absence of an EXTENDED BASE statement, the default allocation is 256.

Each labeled EXTENDED BLOCK is assigned memory independent of each other and is biased at its own unique extended basepage address. An EXTENDED BASE statement cannot be given for BLANK EXTENDED; thus, EXTENDED BLOCK // together with EXTENDED BASE 256 is incorrect.

Syntax - 2 - Vector Processor use only

EXTENDED BASE/VPYn/*

VPYn is the extended block storage name of the data area (n is the number of the bus containing the data area) VPY1, VPY2, VPY3.

13.4 EXTENDED DUMMY Statement

If a subroutine or function is to accept extended memory arguments, those arguments must be declared with the EXTENDED DUMMY statement. If extended arguments are passed to nonextended dummy arguments, unpredictable results will occur.

Syntax

EXTENDED DUMMY u_1, u_2,...,u_i

u_i Dummy arguments representing variable or array names defined in EXTENDED BLOCK storage.

The EXTENDED DUMMY statement must follow FUNCTION and SUBROUTINE statements for subprograms that accept arguments in extended storage.

The following examples show the FORTRAN specification statements for extended memory addressing. The examples accomplish, respectively, zeroing 32K words of extended memory, using the EXTENDED DUMMY statement, initializing arrays in extended memory, and initializing arrays in DATAPOOL. These examples are for a CONCEPT/32 computer; therefore, 2KW map blocks apply.

Example 1

```
C   USING THE EXTENDED BLOCK DEFAULTS,
C   THE FIRST 32K WORDS OF BLANK EXTENDED
C   ARE INITIALIZED TO ZERO
    PARAMETER (NASK=16)
    INTEGER NGET
    EXTENDED BLOCK M(32768)
    CALL X:GDSPCE(NASK,NGET,,)
    DO I=1,32768
      M(I)=0
    END DO
    STOP
    END
```

Example 2

```
C   SET 32K WORDS OF EXTENDED MEMORY
C   INITIALIZE EACH BYTE IN A SUBROUTINE
    PARAMETER (NASK=16)
    EXTENDED BLOCK B
    INTEGER NGET
    INTEGER*1 B(131072)
    CALL X:GDSPCE (NASK,NGET,,)
    CALL SETB(B)
    STOP
    END
    SUBROUTINE SETB(M)
    EXTENDED DUMMY M
    INTEGER*1 M(131072)
    M = 1
    RETURN
    END
```

Example 3

```
C   XX STARTS AT PAGE 256 (80000)
C   YY STARTS AT PAGE 266 (85000)
C   ZZ STARTS AT PAGE 276 (8A000)
    PARAMETER (NASK=12)
    EXTENDED BLOCK/XX/X(1000)/YY/Y(1000)/ZZ/Z(1000)
    EXTENDED BASE/ XX/256/YY/266/ZZ/276
    INTEGER NGET

    CALL X:GDSPCE (NASK,NGET,,)
    DO I=1, 1000
    X(I) = I
    Y(I) = X(I)*I
    Z(I) = X(I)*Y(I)
    END DO
    STOP
    END
```

Example 4

```
REAL*4 X(100), Y(100), Z(100)
EXTENDED BLOCK/DATAPOOL/X,Y,Z
CALL X DPXMNT(ISTAT)
IF (ISTAT .NE. 0) THEN
    PRINT *,'DATAPOOL NOT INCLUDED, STATUS = ',ISTAT
ELSE
    DO I=1,100
       X(I) = I
       Y(I) = X(I)*I
       Z(I) = X(I) + Y(I)
    END DO
END IF
STOP
END
```

13.5 Extended Memory Restrictions

The following are restrictions on the use of extended memory:

- Block storage data in the extended memory area cannot be initialized by FORTRAN 77+ DATA statements, BLOCK DATA subprograms, or type declaration statements. The extended variables must be initialized by the user after the memory has been included by an explicit call to the Scientific Run-Time Library.
- Procedure code execution is not supported in extended memory.
- The variable in an assigned GO TO cannot be in extended memory.
- Variables or arrays that specify formats cannot be specified in extended memory.
- Variables in extended memory cannot be used as arguments in statement functions.
- If the FORTRAN 77+ program includes references to a DATAPOOL in extended memory, both the DATAPOOL partition and the required DATAPOOL dictionary must exist before catalog time of the task.
- If the FORTRAN 77+ program includes references to a global memory partition in extended memory, the partition must exist before catalog time.
- Support for extended memory is provided for FORTRAN users and for assembly language users explicitly triggering the extended addressing mode and explicitly managing all such addressing. The operating system's environment incorporates no conventions for managing or protecting the extended memory area beyond 512KB. Shared use among multiple programs will require user-established conventions for memory management and protection.

- When using a DATAPOOL in extended memory, the largest symbol offset in the DATAPOOL must not exceed the amount of extended memory available on the current machine, i.e., .5MB on a 32/77, 1.5MB on a 32/27 or 32/87, and 15.5MB (minus the operating system) on a 32/67 or 32/97.

CHAPTER 14

INLINE ASSEMBLY LANGUAGE CODING

14.1 Introduction

All lines following an INLINE statement (up to an ENDI statement) are treated as assembly language instructions. The inline code is read by the FORTRAN 77+ compiler. You must save and restore registers as required. There is no optimization across inline code.

Any INLINE statement must follow all specification statements and any statement function statements. Furthermore, FORTRAN 77+ language data definition is not permitted. If the FORTRAN 77+ program unit in which the INLINE statement is found is a function or a subroutine, the necessary linkage code will be generated before the INLINE code is generated.

An assembly line can have a maximum length of 72 characters and is considered to be composed of a maximum of four distinct fields: the label field, operation field, argument field, and comment field. Continuation lines are not permitted.

14.2 Label Field

The label field begins in column 1 and is terminated before column 6. A label, when present, conforms to the requirements of a FORTRAN 77+ statement number and can be preceded by a right parenthesis.

14.3 Operation Field

The operation field begins after column 6 and before column 17. The operation field is mandatory and is terminated by a blank column.

14.4 Argument Field

The argument field begins in the first nonblank column following the operation field and is terminated by a blank column. If column 72 or 10 blank columns are encountered, the argument field is assumed to be blank.

14.5 Comment Field

The comment field begins in the first nonblank column following the argument field and terminates in column 72.

14.6 General Instruction Format

There are five types of general instruction formats:

- Memory reference instructions
- Interregister instructions
- Immediate operand instructions
- Memory bit and condition code instructions
- Operation control instructions

A more detailed explanation of instructions and instruction formats can be found in the appropriate Gould CPU reference manual. All instruction mnemonics supported by the assembler are supported by FORTRAN 77+.

14.6.1 Memory Reference Instructions

Label Field	Operation Field	Argument Field
[label]	instr	reg,[*]addr [,index]

label — Any valid statement number.

instr — The instruction.

reg — The destination or accumulating register.

* — Indicates indirect addressing.

addr — A symbolic or numeric memory address.

index — The index register.

One special case of this type of instruction format is:

[label]	instr	reg,addr

examples of which are the BIB, BIH, BIW, and BID instructions.

Another special case of this type of instruction format is:

[label]	instr	[*] addr [,index]

examples of which are the BU, BL, and BRI instructions.

14.6.2 Interregister Instructions

Label Field	Operation Field	Argument Field
[label]	instr	sreg,dreg

label Any valid statement number.
instr The instruction.
sreg The source or referenced register.
dreg The destination or accumulating register.

14.6.3 Immediate Operand Instructions

Label Field	Operation Field	Argument Field
[label]	instr	reg,value

label Any valid statement number.
instr The instruction.
reg The destination or accumulating register.
value An integer constant in the range -32768 to 32767 (i.e., 2^{-15} to $2^{15}-1$).

One special case of this type of instruction format is:

[label]	instr	reg

examples of which are the EXR, EXRR, LCS, ES, RND, and ZR instructions.

Another special case of this type of instruction format is:

[label]	instr	value

examples of which are the CALM, EI, AI, RI, DAI, DI, and SVC instructions.

14.6.4 Memory Bit and Condition Code Instructions

Label Field	Operation Field	Argument Field
[label]	instr	value, [*] addr [,index]

label Any valid statement number.
instr The instruction.
value A numeric quantity.
* Indicates indirect addressing.
addr A symbolic or numeric memory address.
index The index register.

14.6.5 Operation Control Instructions

Label Field	Operation Field	Argument Field
[label]	instr	

label Any valid statement number.

instr The instruction.

The instructions included in this group are NOP, WAIT, and HALT.

14.7 Argument Field Conventions

All nonaddress items in the argument field must be written as integer constants. When referring to a register, for example, only the digits 0-7 can be used; a reference to an index register must use the digits 1-3.

Address expressions must take one of the following forms:

$ Reference to current location counter value

$$ Absolute zero used as an address.

)s Reference to statement s.

v A variable or array name.

na n is a decimal absolute address, and a (if present) is a B for byte count, H for halfword count, W for word count, or D for doubleword count.

c Any FORTRAN 77+ real or double precision constant. The address of the constant will be used as the address expression.

b±na The parameter na is described above; b is one of the first types of address expressions defined above.

14.8 DATA Directive

Label Field	Operation Field	Argument Field
[label]	DATA [i]	value 1, [value 2 [,...,[value n]. . .]]

The DATA directive causes the constants in the argument field to be output in-line within the object program. DATAi can be of the form DATAB, DATAH, DATAW, and DATAD, representing byte, halfword, word, and doubleword data fields, respectively. DATA is interpreted as DATAB. The values in the argument field can be FORTRAN 77+ constants or literal strings. This directive automatically aligns the specified data to the proper storage boundary.

14.9 GEN Directive

Label Field	Operation Field	Argument Field
[label]	GEN	field list

The GEN directive is used to assemble data of different types and sizes into a string of contiguous bits. Each subfield contains a positive integer representing the subfield size in bits, followed by a (/) character, followed by a FORTRAN 77+ constant or variable. Each subfield is packed in contiguous bits starting from the leftmost byte. If the total length of all subfields is not a multiple of eight bits, the last byte will be padded with trailing zeros. All address fields must be at least 20 bits in length and right-justified within a word.

Example

```
GEN     6/1,6/-1,20/I
GEN     3/7
```

The first line in the above example assembles to 07F00000 (hexadecimal) + address of I. The second line produces a byte E0 (hexadecimal).

14.10 AC Directive

Label Field	Operation Field	Argument Field
[label]	AC [i]	[*] addr [,index]

The AC directive is used to generate an address with indexing and/or indirect addressing. It can also be used to reserve a 32-bit zero on a word boundary. ACi can be of the form ACB, ACH, ACW, and ACD, representing byte, halfword, word, and doubleword address fields, respectively. AC will be interpreted as ACB. This directive always forces assembly to a word boundary. Bits 0 through 8 of the generated word are always set to zero.

14.11 BOUND Directive

Label Field	Operation Field	Argument Field
[label]	BOUND	boundary

The BOUND directive advances the location counter so that it is a byte multiple of the boundary designated. boundary is any of the positive integers 1, 2, 4, 8, 16, or 32.

14.12 RES Directive

Label Field	Operation Field	Argument Field
[label]	RES	n [a]

The RES directive causes the location counter to be advanced n times of size a. n is a positive integer constant designating the number of units to be reserved. If a is omitted, n is interpreted as a byte count. a can be B for byte reservation, H for halfword reservation, W for word reservation, and D for doubleword reservation. Halfword, word, and doubleword reservations will be forced to the correct boundary.

14.13 EQU Directive

Label Field	Operation Field	Argument Field
symbol	EQU	integer constant

The EQU directive allows the user to equate any valid symbol used as a label to the integer constant as an absolute logical address value. The symbol in the label field is mandatory and cannot be redefined or match any FORTRAN 77+ variables. Reference the symbol only in the address field of a memory reference instruction or as the argument of an AC directive. The address value of this symbol has byte attribute unless the context of the reference indicates otherwise. (A symbol beginning with letters C or X in column 1 results in the line being considered a comment line.)

14.14 Referencing Variables in Local Storage, COMMON, GLOBAL COMMON, or DATAPOOL

Variables in local memory, COMMON, GLOBAL COMMON, or DATAPOOL can be referenced in INLINE assembly code by referring to the item by name. For example:

```
INLINE
LW  3,I
STW 3,J
ENDI
```

is equivalent to the FORTRAN 77+ statement J=I.

14.15 Referencing Dummy Variables

Reference dummy variables, except extended dummies, as indirect. For example in:

```
SUBROUTINE SUB(A,B)
```

the sequence:

```
INLINE
   LW 4,*A
   STW 4,*B
   ENDI
```

would be equivalent to the FORTRAN 77+ statement B=A.

To reference an extended dummy variable, load its address (by referring to the variable name) into an index register, then reference the item by indexing with a zero offset. For example, in:

```
SUBROUTINE EXSUB (X,Y)
EXTENDED DUMMY X,Y
```

the sequence:

```
INLINE
   LW 3,X
   LW 2,Y
   SEA
   LW 7,0,3
   STW 7,0,2
CEA
ENDI
```

would be equivalent to the FORTRAN 77+ statement Y=X.

14.16 Referencing Variables in Extended Memory

To reference an item in extended memory, first load an index register with the base of the extended block, then index each reference to the variable with its extended block base. Given the data structure:

```
EXTENDED BLOCK  /IBLOCK/I1,I2
EXTENDED BASE  /IBLOCK/256
EXTENDED BLOCK /KBLOCK/K1,K2
EXTENDED BASE /KBLOCK/257
INTEGER  IBLOCKAD, KBLOCKAD

IBLOCKAD  =  256*2048
KBLOCKAD  =  257*2048
```

the sequence:

```
INLINE
   SEA
   LW 1, IBLOCKAD
   LW 2, KBLOCKAD
   LW 3, I1,1
   STW 3, K1,2
   CEA
   ENDI
```

would be equivalent to the FORTRAN 77+ statement K1=I1.

14.17 Setting and Clearing Extended Addressing Mode

The user is responsible for setting extended addressing (SEA) mode or clearing extended addressing (CEA) mode in any inline coding. The user is also responsible for making sure that the addressing mode in force by the compiler when inline code is entered is still in force when exiting inline code.

CHAPTER 15

USING THE FORTRAN 77+ COMPILER

15.1 Introduction

This chapter describes how to use options, job control language, devices, and assembler language routines with FORTRAN 77+ programs. It also describes how to call FORTRAN 77+ subroutines from assembler language programs.

Refer to Appendix B for examples.

15.2 Logical File Code Assignments

The following are the logical file codes (LFC) associated with the FORTRAN 77+ compiler and their default assignments.

LFC	Default	Function
SI	$AS SI TO SYC	source input
LO	$AS LO TO SLO	listed output
BO	$AS BO TO SBO	binary object output
GO	$AS GO TO SGO	binary object output
*U1	$AS *U1 TO TEMP SIZE=250 BLOC=Y	temporary utility file
*U2	$AS *U2 TO TEMP SIZE=250 BLOC=Y	temporary utility file
*U3	$AS *U3 TO TEMP SIZE=100 BLOC=N	temporary utility file (for cross reference)
*U4	dynamically allocated by FORTRAN 77+	input for INCLUDE directive files

15.3 Compiler Options

The following FORTRAN 77+ compiler options can be specified in an $OPTION statement using job control, or they can be specified or changed within a program using the FORTRAN 77+ OPTION directive (refer to Chapter 2).

Option	Result
1	Suppresses listed output on logical file code LO.
2	Suppresses binary output to SBO file on logical file code BO.
3	Suppresses storage dictionary on logical file code LO.
4	Suppresses symbol cross reference on logical file code LO.
5	Enables general object output on logical file code GO.
6	Lists generated code on logical file code LO.

Option	Result
7	Allows mismatched argument lists when passing arguments to subprograms. Causes arguments to be processed using F.PR instead of being processed inline. For more information refer to processing of arguments for subprogram calls in Chapter 9.
8	Suppresses the flagging of duplicate type declarations as errors.
9	Compiles source records containing a Y in column 1 as if the Y were a blank. If this option is not set, these records are treated as comments.
10	Character constant actual arguments are treated as Hollerith constants instead of character constants when passed to a subprogram.
11	Code for DO loops is generated to execute all DO loops at least once.
12	Enables the compiler to run in compatible mode even though native mode was selected at installation.
13	Enables the generation of subroutine calls for real-to-integer and integer-to-real conversions (non-hardware assisted) even though the hardware FIX and FLOAT options (hardware assisted) were selected at installation.
14	Utilize the CONCEPT 32/67 Scientific Accelerator for math intrinsic functions ALOG, ALOG10, SIN, COS, ATAN, SQRT, and EXP.
15	Includes compilation date and time in object modules (refer to Chapter 2).
16	Utilize the operating system to find big blocking buffers.
17	Enables the compiler to run in native mode even though compatible mode was selected at installation.
19	Outputs symbolic table information for use by the symbolic debugger.
20	Compiles source records containing an X in column 1 as if the X were a blank. If this option is not set, these records are treated as comments.

Rules for Use

- Options 1, 2, 3, and 4 may be changed within a program only if they have not been set in a $OPTION job control statement.
- Options 7, 8, 11, 12, 13, and 16 may not be changed within a program.
- Options 6, 10, and 15 may be changed within a program regardless of their having been set or not set in a $OPTION job control statement.
- Options 9 and 20 can be changed within a program only if it has been set in a $OPTION job control statement.

- Changes to options 1, 4, 6, 9, 10, and 20 will take effect at the point in the program where they occur and will remain in effect until changed by another OPTION directive.

- Option 2 can be changed only before any declarations and all executable statements in a program unit. Option 2 can be changed before or after a PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statement.

- Options 3, 15 and 19 will be considered on or off for an entire program unit according to the last OPTION (or $OPTION job control statement) directive seen before the end of the program unit.

15.4 Run-Time Options

Option	Result
7	Minor errors that might abort a program are ignored (refer to Appendix D for a list of minor errors).
8	The default for the BLANK specifier becomes zero for formatted input.

Options 7 and 8 may not be changed within a program by means of the FORTRAN OPTION statement.

15.5 Job Control Language

Job control language (JCL) specifies the desired run-time environment. Some of the most common JCL statements used with FORTRAN 77+ programs are listed below (refer to the MPX-32 Reference Manual for a complete description).

$JOB jobname	Specifies the start of a batch job.
$EOJ $$	Specifies the end of a batch job.
$OPTION op1,op2,...,opn	Specifies the options for the next batch program to be executed.
$RUN program	Executes a cataloged program in current working directory.
$EXECUTE program	Executes a cataloged program in system directory.
$ALLOCATE size	Allocates the memory size (in bytes) needed for program execution. (The FORTRAN 77+ compiler dynamically allocates its required memory. Therefore, the $ALLOCATE statement should not be used for compiler execution.)
$ASSIGN lfc TO file or $ASSIGN lfc TO DEV= device or $ASSIGN lfc TO LFC= alternate lfc	Assigns a file, device, or another logical file code to the logical file code (unit) used in the next program to be executed for I/O.

Do not code the character $ in column 1 of the source program or input data passing through the SYC stream. Doing so causes premature termination of compilation or execution with unexpected results.

15.5.1 Compiling

The following is an example of the JCL needed to compile a FORTRAN 77+ program under MPX-32.

```
$JOB COMPILE TEST1 SLOF=E.SORT
$ASSIGN SI TO S.TSORT
$EXECUTE FORT77
$EOJ
$$
```

15.5.2 Compiling, Cataloging, and Executing

The following is an example of the JCL needed to compile, catalog, and execute a FORTRAN 77+ program under MPX-32.

```
$JOB COMPILE YOUNG SLOF=E.SORT
$NOTE COMPILE CATALOG AND EXECUTE A PROGRAM
$ASSIGN SI TO S.TSORT
$ASSIGN BO TO O.TSORT
$ASSIGN LO TO L.TSORT
$EXECUTE FORT77
$ALLOCATE 10000
$ASSIGN SGO TO O.TSORT
$EXECUTE CATALOG
ASSIGN SI TO SYC
ASSIGN LO TO SLO
BUILD WORDSORT
$RUN WORDSORT
0010
MANIC
SEVERAL
FULFILL
COMIC
SLATE
REVISION
MOOD
APPLE
STALWART
REFLECT
$EOJ
$$
```

NOTE:

$ASSIGN SI TO S.TSORT	Assigns the logical file code SI to the source input file S.TSORT.
$ASSIGN BO TO O.TSORT	Assigns the logical file code BO to the object file O.TSORT.

$ASSIGN LO TO L.TSORT	Assigns the logical file code LO to the output file L.TSORT.
$ASSIGN SGO TO O.TSORT	Assigns the logical file code SGO to the object file O.TSORT for the cataloger.
$ALLOCATE 10000	Provides adequate core space for the loader and the execution of the program.
ASSIGN SI TO SYC	Assigns the program's logical file code SI to the SYC file.
ASSIGN LO TO SLO	Assigns the program's logical file code LO to the SLO file.
$RUN WORDSORT	Loads and executes the program WORDSORT.
0010...REFLECT	Data for the program.

15.5.3 Compiling and Cataloging

The following is an example of the JCL needed to compile and catalog a FORTRAN 77+ program for future execution. This job involves two steps:

1. $EXECUTE FORT77	Compiles the program and places the resulting object code in the SGO file.
2. $EXECUTE CATALOG	Catalogs the object code from the SGO file and places it into the newly created disc file.

```
$JOB COMP&CAT TEST1 SLOF=E.SORT
$NOTE COMPILE AND CATALOG A PROGRAM
$OPTION 5
$EXECUTE FORT77
  .
  .(Note: FORTRAN source statements go here)
  .
$EXECUTE CATALOG
ASSIGN SI TO SYC
ASSIGN LO TO SLO
BUILD WORDSORT
$EOJ
$$
```

15.6 Using a Tape File as a Data Source

The following sequence uses the library format tape file SORTDATA. (A previously cataloged FORTRAN 77+ program uses logical file code SI for input.)

```
$JOB TAPE.IN
$NOTE READS WORDS FROM A TAPE FILE 'SORTDATA'
$OPTION 8
$ASSIGN SI1 TO DEV=MT BLOCKED=N
```

```
$EXECUTE UPDATE
/SKIP SORTDATA
/EXIT
$ASSIGN SI TO DEV=MT BLOCKED=N
$RUN WORDSORT
$EOJ
$$
```

Note:

$OPTION 8	Option 8 specifies that the SI1 file will be unblocked.
$ASSIGN SI1 TO DEV=MT BLOCKED=N	Assigns the logical file code SI1 to a magnetic tape drive. The operator's console will receive a message requesting that the tape be mounted when this job is executed.
$EXECUTE UPDATE /SKIP SORTDATA /EXIT	Advances the tape to the file with header SORTDATA.
$ASSIGN SI TO DEV=MT BLOCKED=N	Assigns the logical file code SI (from the cataloged FORTRAN program) to the tape drive, thus overriding the cataloged assignment for SI.
$RUN WORDSORT	Executes the cataloged program.

15.7 Using a Disc File for Output Data

The following example places the output of WORDSORT in the disc file OUTSORT. WORDSORT uses logical file code (unit) LO for output.

```
$JOB DISK.OUT OWNRNAME
$NOTE WRITES SORTED WORDS TO USER DISC FILE 'OUTSORT'
$EXECUTE VOLMGR
CREATE FILE OUTSORT
$ASSIGN LO TO OUTSORT
$RUN WORDSORT
0010
MANIC
SEVERAL
FULFILL
COMIC
SLATE
REVISION
MOOD
APPLE
STALWART
REFLECT
$EOJ
$$
```

Note:

$EXECUTE VOLMGR	Executes the volume manager.
CREATE FILE OUTSORT	Creates the user file OUTSORT in the user's current directory.
$ASSIGN LO TO OUTSORT	Assigns the logical file code LO of the cataloged program to the disc file OUTSORT.
$RUN WORDSORT	Executes the cataloged program WORDSORT.

15.8 Using Data from Cards

The following is an example of using the card reader as an input device. The FORTRAN program WORDSORT uses logical file code (unit) SI for input.

```
$JOB CARD.IN OWNRNAME
$NOTE READS WORDS FROM CARD READER
$ASSIGN SI TO DEV=CR
$RUN WORDSORT
$EOJ
$$
```

Note:

$ASSIGN SI TO DEV=CR	Assigns the logical file code SI from the cataloged program to the card reader. When this job is run, the operator's console will print a message asking for the cards to be placed in the reader. The user must place an EOF card (2-3-4-5 multi-punch in column 1) behind the last card.
$RUN WORDSORT	Executes the cataloged program.

15.9 Calling Assembly Routines from FORTRAN 77+ Programs

The FORTRAN 77+ compiler generates three different types of calling protocols for CALL statements. The code generated is determined by the number of parameters in the CALL statement. The three types of calling protocols are associated with the following:

- Calling the assembler routine with no parameters.
- Calling the assembler routine with one parameter.
- Calling the assembler routine with more than one parameter.

The code generated must follow protocols for:

- Parameter locations (addresses)
- Return address

When parameters are passed by FORTRAN 77+, the parameters are formatted as shown in Table 15-1.

15.9.1 Assembly Routine with No Parameters

To be called by a FORTRAN 77+ routine, the assembly routine must:

- Declare the name of the assembler entry point (DEF directive).
- Return to the FORTRAN 77+ routine by a transfer register to the program status word instruction (TRSW) using the content of Register 0 upon entry to the routine. Alternatively, the return address can be stored in a memory location and return is effected by an unconditional branch indirect (BU * RETADDR).

15.9.2 Assembly Routine with One Parameter

For a FORTRAN 77+ program to call an assembler routine with one parameter, the assembler routine must:

- Declare the name of the assembler entry point (DEF directive).
- Use the value in Register 1 (one) as the address of the single parameter from the FORTRAN 77+ program that called the routine.
- Return to the FORTRAN 77+ routine by means of the link address in Register 0.

15.9.3 Assembly Routine with Two or More Parameters

For a FORTRAN 77+ routine to call an assembler routine with more than one parameter, the assembler routine must:

- Declare the name of the assembler entry point (DEF directive).
- Use the parameter area in the FORTRAN 77+ routine pointed to by the address in Register 0, as illustrated in the following example.
- Calculate the return address and transfer to it (as illustrated in the following example).

15.9.4 Parameter Area

Example

```
word 0      number of parameters*4 (pointed to by register 0)
word 1      address, possibly indirect, of parameter one
word 2      address, possibly indirect, of parameter two
                          :
                          :
word n      address, possibly indirect, of last parameter
```

Each of the entries in the parameter area is one word long on a word boundary.

15.9.5 Calculation of the Return Address

The assembler routine can calculate the address within the FORTRAN 77+ routine to which control should return. This address is that of the first word after the parameter area. If the PSW (received upon call of the assembler routine) is still in Register 0, then the following assembler code will update Register 0 and return to the correct return address:

```
TRR     R0,R1
ABR     R0,29
ADMW    R0,0,R1
TRSW    R0
```

Example

The following sequence shows a FORTRAN 77+ routine calling an assembler program to print the addresses of specific 'C.DOTS'.

CALLER

```
1                       PROGRAM CALLER
2               C       *-----------------------------------------------------------------
3               C       *       THIS PROGRAM IS AN EXAMPLE OF A FORTRAN PROGRAM CALLING
4               C       *  AN ASSEMBLER PROGRAM TO PRINT THE ADDRESSES OF SPECIFIC
5               C       *  'C.DOTS'.
6               C       *-----------------------------------------------------------------
7               C
8                       INTEGER     NAME,VALUE,BLANK
9                       DATA        BLANK /4H    /
10              C
11              C       LOOP:  GET NAME, CALL CDOT, PRINT RESULTS
12              C
13                      DO                                          ! DO FOREVER
14              C
15                          READ (*,FMT='(A4)') NAME                ! READ IN A NAME
16                          IF (NAME.EQ.BLANK) STOP                 ! BLANK INDICATES LAST INPUT
17                          CALL CDOT(NAME,VALUE)                   ! SEARCH FOR 'C.'
18                          IF (VALUE.EQ.0) THEN                    ! IF ZERO : NOT FOUND
19                              WRITE (*,70) NAME                   ! WRITE NAME ONLY
20                          ELSE                                    !      ELSE
21                              WRITE (*,71) NAME,VALUE             ! WRITE NAME AND ADDRESS
22                          ENDIF
23              C
24                      ENDDO
25              C
26              C       LABELED FORMAT STATEMENTS
27              C
28              70      FORMAT (1X,'''C.',A4,''' DOES NOT EXIST')
29              71      FORMAT (1X,'C.',A4,'=',Z8)
30                      END
```

```
00007                                   PROGRAM CDOT
00008                                   DEF     CDOT
00009                           ********************************************************************
00010                           *       CDOT - RETURNS VALUE (ADDRESS) OF A C.XXXX
00011                           *
00012                           *       NOTE:  ALL CDOT NAMES MUST BE FOUR ASCII CHARACTERS, LEFT-
00013                           *              JUSTIFIED, BLANK FILLED.  THE ADDRESS OF THE VARIABLE
00014                           *              CONTAINING THE NAME MUST BE ON A WORD BOUNDARY.  THIS
00015                           *              SUBROUTINE IS FORTRAN COMPATABLE.
00016                           ********************************************************************
00017  P00000                           M.EQUS
                                        ENDM
00018                           *
00019                           *       NAMES OF 4-CHARACTER CDOTS
00020                           *
00021  P00000                   NAMES   EQU     $
00022  P00000  41464C57                DATAW   C'AFLW'
00023  P00004  414E5957                DATAW   C'ANYW'
00024  P00008  42495420                DATAW   C'BIT '
00025  P0000C  43454E54                DATAW   C'CENT'
00026  P00010  43555252                DATAW   C'CURR'
00027  P00014  44415445                DATAW   C'DATE'
00028  P00018  44415920                DATAW   C'DAY '
00029  P0001C  44515545                DATAW   C'DQUE'
00030  P00020  46524545                DATAW   C'FREE'
00031  P00024  484F4C44                DATAW   C'HOLD'
00032  P00028  4D49444C                DATAW   C'MIDL'
00033  P0002C  504F4F4C                DATAW   C'POOL'
00034  P00030                   ENDNAMES EQU    $
00035                           *
00036                           *       VALUES OF 4-CHAR CDOTS IN SAME ORDER AS NAMES OF 4-CHAR CDOTS
00037                           *
00038  P00030                   VALUES  EQU     $
00039  P00030  0008000E  0000E         ACB     C.AFLW
00040  P00034  000809D8  009D8         ACB     C.ANYW
00041  P00038  00080A8C  00A8C         ACB     C.BIT
00042  P0003C  00080808  00808         ACB     C.CENT
00043  P00040  000808E8  008E8         ACB     C.CURR
00044  P00044  00080800  00800         ACB     C.DATE
00045  P00048  00080808  00808         ACB     C.DAY
00046  P0004C  00080AA4  00AA4         ACB     C.DQUE
00047  P00050  00080800  008D0         ACB     C.FREE
00048  P00054  000809CC  009CC         ACB     C.HOLD
00049  P00058  00080AD0  00AD0         ACB     C.MIDL
00050  P0005C  00080AF4  00AF4         ACB     C.POOL
00051  P00060                   ENDVAL  EQU     $
```

```
CDOT

00053                                   *
00054                                   *       EXECUTABLE CODE
00055                                   *
00056   P00060  DC0000C0  P000C0  CDOT   STF     R0,SAVEREGS
00057   P00064  2C80                    TRR     R0,R1
        P00066       0002
00058   P00068  35300004   00004         LA      R2,*1W,R1       ACTUAL ADDRESS OF FIRST PARM(NAME)
00059   P0006C  AF400000   00000         LW      R6,0W,R2        LOAD CDOT NAME
00060                                   *
00061                                   *       FIND MATCH OF 4-CHAR NAME TO GET OFFSET INTO VALUE TABLE .
00062                                   *
00063   P00070  CA80FFD0          FOURC  LI      R5,NAMES-ENDNAMES  GET NEGATIVE # WORDS IN NAME TABLE
00064   P00074  0DR0                    ZR      R3                 BIAS FOR LOOP
        P00076       0002
00065   P00078  93600000  P00000 LOOP4  CAMW    R6,NAMES,R3     DO NAMES MATCH ?
00066   P0007C  EE000091  P00090         BEQ     MATCH4          IF SO - BRANCH
00067   P00080  22R3                    ABR     R3,29           ELSE - INCREMENT BIAS
        P00082       0002
00068   P00084  F6C00079  P00078         BIW     R5,LOOP4        INCREMENT AND TRY AGAIN UNLESS ZERO
00069                                   *
00070                                   *       NAMES DON'T MATCH
00071                                   *
00072   P00088  D6B00008   00008         STW     R5,*2W,R1       STORE ZERO USING PARAMETER ADDRESS
00073   P0008C  EC000090  P0009C         BU      BYE             GO RETURN TO CALLER
00074                                   *
00075                                   *       GET VALUE OF 4-CHAR NAME AND STORE AS PARM TO CALLER
00076                                   *
00077   P00090  AEE00030  P00030 MATCH4  LW      R5,VALUES,R3    GET VALUE USING OFFSET
00078   P00094  1E51                    ZBR     R5,12           ZERO BYTE-ADDRESS-INDICATOR BIT
        P00096       0002
00079   P00098  D6R00008   00008         STW     R5,*2W,R1       STORE VALUE USING PARAMETER ADDRESS
00080                                   *
00081   P0009C  BA200003   00002  BYE    ADMH    R0,1H,R1        ADD PARAMETER COUNT BIAS
00082   P000A0  C8010004          ADI     R0,1W           ADD ONE WORD BIAS
00083   P000A4  D40000C0  P000C0  STW     R0,SAVEREGS     SAVE RETURN ADDRESS
00084   P000A8  CC0000C0  P000C0  LF      R0,SAVEREGS     RESTORE REGISTER CONTEXT
00085   P000AC  2800              TRSW    R0              RETURN TO CALLING PROGRAM
00086                                   *
00087   P000C0                    BOUND   1F
00088   P000C0                SAVEREGS RES  1F
00089   P000E0                    END
*  0000  ERRORS IN CDOT
$EOJ
```

15.9.6 Function Calling Conventions

Arguments to the function are passed just as they are for subroutines. The function value (if not of CHARACTER data type) is returned in one or more registers as follows:

Function Type	Value Returned in Register(s) Number	
INTEGER*1,*2,*4	7	(value is right-justified in R7)
INTEGER*8	6,7	
REAL*4	7	
REAL*8,COMPLEX*8	6,7	
COMPLEX*16	4,5,6,7	
LOGICAL*1,*4	7	

A CHARACTER type function value is returned in a memory area, the address of which is supplied with the calling parameter list. This memory area and its address are established by the compiler transparently to the user.

15.10 Calling FORTRAN 77+ Subroutines from Assembly Language Programs

The FORTRAN 77+ compiler generates three different types of calling protocols for CALL statements. The code generated is determined by the number of parameters. Thus, for an assembler program to call a FORTRAN 77+ program, the assembler program must use the same calling protocol as the FORTRAN 77+ program.

Function and subroutine calling conventions are the same, except the function value is returned in one or more registers.

15.10.1 FORTRAN 77+ Subroutine with No Parameters

To call a FORTRAN subroutine with no parameters:

1. Declare the subroutine name external (MACRO statement EXT).
2. Branch and link to the subroutine (MACRO operation BL).

The FORTRAN 77+ subroutine will return control to the assembler routine at the word following the branch and link.

15.10.2 FORTRAN 77+ Subroutine with One Parameter

To call a FORTRAN 77+ subroutine with one parameter:

1. Declare the subroutine name external (MACRO statement EXT).
2. Load Register 1 (one) with the parameter address.
3. Branch and link to the subroutine (MACRO operation BL).

The FORTRAN 77+ subroutine will return control to the assembler routine at the word following the branch and link.

Start the parameter storage location in the assembler routine on a boundary compatible with the type of FORTRAN 77+ variable associated with the parameter.

15.10.3 FORTRAN 77+ Subroutine with Two or More Parameters

To call a FORTRAN 77+ subroutine with two or more parameters:

1. Declare the subroutine name external (MACRO statement EXT).
2. Set up a parameter area immediately after the branch and link (refer to the following examples).
3. Branch and link to the FORTRAN 77+ subroutine (MACRO operation BL).

Example

PARAMETER AREA

```
BL          FORTRAN subroutine name
DATAW nW    number of parameters
ACx         address of first parameter
ACx         address of second parameter
.
.
ACx         address of last parameter
```

15.10.4 Parameter Lists Generated by the Compiler

The preceding information is adequate for normal FORTRAN 77+ subprogram calls where extended memory parameters are not involved. For other cases, it may be necessary to duplicate the full parameter word content as generated by the compiler; this content includes parameter type information with descriptor flags for array and for extended memory parameters. Table 15-1 describes the full parameter word content.

Table 15-1
Compiler Parameter Lists

Bit(s)	Usage
0	A pseudo-indirect flag indicating that a parameter word points to a remote parameter word containing the 24-bit address of an extended memory entity.
1*	An extended memory flag indicating that a parameter word has a 24-bit address field.
2*	A pseudo-F-bit flag (when bit 1 is set) indicating that a byte is addressed.
3*	A bit parameter flag (when bit 1 is set) indicating that a specific entity is of type bit.
4-7	A type code (when bit 3 is reset), as explained in Table 15-2. or A bit position within a byte (when bits 1, 2, and 3 are set).
8	An array flag indicating the parameter is an array (used by argument transfer routines A.TF and A.TU).
9-10	Unused.
11	An indirect flag indicating the parameter address is to be obtained by indirect reference.
12-31**	An address field, including format bits for halfword or doubleword entities.

* Bits 1, 2, and 3 are only present in a remote parameter word pointed to by an actual parameter list word in which bit 0 is set.

** When the format bits are set and indexing takes place, the format bits will alter the instruction. To prevent this, either remove the format bits before the address is used or place the address into a storage location and access the parameter value indirectly through that location. However, the indirect method must not be used for extended addresses.

Table 15-2
Type Code

Type Code (Hexadecimal)	Type
0	INTEGER*1
1	INTEGER*2
2	INTEGER*4
3	INTEGER*8
4	REAL*4
5	REAL*8
6	COMPLEX*8
7	COMPLEX*16
8	BIT
9	LOGICAL*1
A	LOGICAL*4
B	CHARACTER

APPENDIX A

I/O USING MPX-32

A.1 Input/Output Terms

The following is a list of FORTRAN 77+ and MPX-32 terms that relate to input/output.

- Formatted - A FORTRAN 77+ term applied to those records read or written under the editing direction of format, e.g., READ (6,99).

- Unformatted - A FORTRAN 77+ term applied to those records read or written without format editing, e.g., READ. Note that the Scientific Run-Time Library applies its own blocking scheme to the physical transfer and this blocking scheme is distinct from MPX-32 file blocking.

- Asynchronous input/output (BUFFERIN/BUFFEROUT) - A FORTRAN 77+ term that describes a method of performing a read or write directly to or from a user's buffer by use of CALL BUFFERIN or CALL BUFFEROUT, respectively. No formatting or padding is done. No-wait input/output is used; the user is responsible for insuring input/output completion before buffer content is used or altered.

- Blocking - An MPX-32 term that describes the MPX-32 Input/Output Control System capability of placing multiple logical records into a 768-byte physical record, along with record control information. Applicable to all spooled files and optionally to disc or magnetic tape user files. Independent of formatted, unformatted, and asynchronous I/O.

- Compressed - An MPX-32 term meaning a format for source information that removes superfluous space codes and replaces them with count control bytes. It is supported by assembler, source update, and editor. Independent of blocking.

- No-Wait I/O - The MPX-32 capability to start (or merely put into queue) an input/output request and to immediately return to the requesting task.

- Direct I/O - The MPX-32 capability of permitting a privileged task to issue its own actual machine language input/output instructions. (Unrelated to the term direct-access input/output.)

- Logical File Code - An MPX-32 term meaning the one- to three-character alphanumeric field used in all file assignments. With the exception of logical file code 'UT', this field has absolutely no fixed system-wide relationship to the permanent file name, the system file, the physical device assignment, and another logical file code.

Logical file code 'UT' is automatically connected to user terminal by the operating system for interactive tasks.

A.2 General Observations

- If one writes a file in a particular combination of method and format, then one generally must read it using the same method and format.
- Some of the methods and formats can be combined.

Formatted and unblocked	Formatted and blocked
Unformatted and unblocked	Unformatted and blocked
Asynchronous and unblocked	Asynchronous and blocked

- Some methods are generally more time-efficient than others:

  Asynchronous more than synchronous
  Unformatted more than formatted
  Unblocked more than blocked

- Blocking is requested by using the OPEN statement with BLOCKED = .TRUE.. If too many (31) blocked file codes are opened, an 'RT99' abort code is issued.
- In MPX-32, assembly language programmers can specify random I/O to 192 words in length by setting bit 4 of word 2 of the file control block (refer to the MPX-32 Reference Manual). This method is also available as nonsupported FORTRAN-callable subroutines in the SORT package, ISAM package, and in the user's group library.

A.3 End-of-File Detection

End-of-file detection is performed for all blocked and/or unformatted files. End-of-file detection is also performed for formatted/unblocked disc and magnetic tape files, card and paper tape reads. In the default detection method, used by the FORTRAN 77+ compiler, X'0F' detected as the first byte of a formatted, unblocked record means end-of-file.

Because of possible conflicts between data patterns and the end-of-file code on formatted/unblocked magnetic tape and disc files, an optional end-of-file method is available. A call to the SRTL routine X:MPXEOF causes the optional method of end-of-file detection to override the default method. In the optional method X'0FE0FE0F' indicates end-of-file when detected as the first word of a record.

This optional method provides a better means of discriminating between data patterns and EOF indicators, but does not completely eliminate ambiguities. If closer discrimination is necessary, blocked files should be used.

A.4 Maximum Sizes

MPX-32 assumed disc sector size: 192 words (768 bytes)

MPX-32 assumed maximum record lengths:

Magnetic tape	8190 bytes
Cards	120 bytes
Line printer	144 bytes
Teletypewriter or CRT	80 bytes

Formatted I/O logical record lengths

Unblocked files	768 bytes
Blocked files	254 bytes
SBO	80 bytes
SGO	254 bytes
SLO	133 bytes
SYC	80 bytes

Unformatted I/O logical record lengths

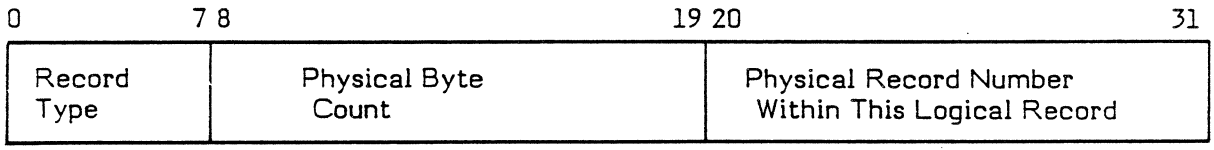
Unblocked files	3,128,580 bytes* (4095 physical records of 192 words: 191 words of user data plus 1 header word)
Blocked files	736 bytes excluding all header words
SBO	116 bytes excluding header word
SGO	116 bytes excluding header word
SLO	250 bytes excluding header word
SYC	116 bytes excluding header word

* Record lengths can be further restricted by physical device characteristics.

A.5 Unformatted I/O Records

For unformatted I/O records, the first word of each physical record is a control word with the following format:

Bits



The physical byte count includes the header record itself.

Record Type (Hexadecimal)	Description
FF	Indicates unformatted record
DF	Indicates last physical record in the logical record
0F	Indicates end-of-file for run-time routines

This information pertains to magnetic tape as well as to disc units.

Unblocked physical records contain 192 words. For blocked files, each physical record is equal to one blocked record. The physical byte count for a blocked record cannot exceed 255 bytes. Logical records can span multiple sectors, while physical records within the logical record cannot.

A.6 Device Type Codes

The following are the device type codes used for the FORTRAN 77+ OPEN statement, INQUIRE statement and the callable MPX-32 subroutines.

Device Type Code (Hex)	Decimal Equivalent	Two-Character Device Mnemonic	Device Description
00	00	CT	Operator console (not assignable)
01	01	DC	Any disc unit
02	02	DM	Any moving-head disc
03	03	DF	Any fixed-head disc
04	04	MT	Any magnetic tape unit
05	05	M9	Any nine-track magnetic tape unit
06	06	M7	Any seven-track magnetic tape unit
07	07	CD	Any card reader or card reader/punch
08	08	CR	Any card reader
09	09	CP	Any card punch
0A	10	LP	Any line printer
0B	11	PT	Any paper tape reader/punch
0C	12	TY	Any teletypewriter (other than console)
0D	13	CT	Operator console (assignable)
0E	14	FL	Floppy disc
0F	15	NU	Null device
10	16	CA	Communications adapter (binary synchronous/asynchronous)
11	17	U0	For user-defined application
12	18	U1	For user-defined application
13	19	U2	For user-defined application
14	20	U3	For user-defined application
15	21	U4	For user-defined application
16	22	U5	For user-defined application
17	23	U6	For user-defined application
18	24	U7	For user-defined application
19	25	U8	For user-defined application
1A	26	U9	For user-defined application
1B	27	LF	Lineprinter/floppy controller (used only with SYSGEN)

APPENDIX B

LISTING EXAMPLES

B.1 Source Listing

The FORTRAN 77+ source listing facility provides a sequenced listing of the 80 byte program source input records and generated statement markers for use with the Symbolic Debugger. Specifying $OPTION 1 or FORTRAN 77+ directive OPTION 1+ suppresses this listing.

B.1.1 Source Listing Format

The source listing begins with the name of the source module, whether it is a program, subroutine, function, etc., in the upper left hand corner of the listing. If listed output (LO) is assigned to system listed output (SLO), the module name appears in the upper left hand corner of every page in the listing.

The listing of the 80 byte input source records appears below the name. The format for each line on the page is as follows:

- A compiler generated line sequence number appears at the left margin.
- A statement marker for use with the Symbolic Debugger under MPX-32, if applicable, appears to the right of the sequence number. The statement markers have the form S.n where n is a compiler assigned statement number. The numbers are always in ascending order, however, they may not always appear in consecutive order since some source statements contain multiple statement constructs. In that case, only the first marker for that line appears in the listing.
- An 80 byte source input record, displayed as read from the assigned inputs for the FORTRAN 77+ compiler, appears several spaces to the right of the statement marker.

This format is maintained throughout the compilation of the entire program unit, unless the generated code listing is also selected. In that case, the two listings will appear to be interspersed.

A source listing is provided for a program, EXAMPLE. This program is not meant to be a program having any utility or to suggest any recommended programming practices. It was written to show a few FORTRAN constructs and how the elements of the constructs relate between the various listing formats. The comments appearing on the right side of the source listing lines have been added to aid in comparison of the other listing formats. They are not generated by the compiler. Also, the relative addresses will change between versions of the compiler. It is important to be able to relate the various listing elements for a given compiler.

The source listing should be compared with the storage dictionary in section B.2, the symbolic cross-reference in section B.3, and the generated code listing in section B.4.

```
1                             PROGRAM EXAMPLE                                                  0004.000
2                                                                                              0005.000
3                           C----- THIS EXAMPLE PROGRAM WAS DESIGNED AS AN AID TO UNDERSTANDING 0006.000
4                           C----- THE WAYS IN WHICH THE SOURCE LISTING, THE STORAGE DICTIONARY, 0007.000
5                           C----- THE SYMBOLIC CROSS REFERENCE, AND THE OPTION 6 GENERATED CODE 0008.000
6                           C----- LISTING RELATE TO EACH OTHER.  FOR CLARIFICATION, THE COMMENTS 0009.000
7                           C----- HAVE BEEN ADDED ON THE STATEMENT LINES TO PROVIDE A MEANS OF   0010.000
8                           C----- VERIFYING RELATED ITEMS.                                      0011.000
9                                                                                              0012.000
10                          C-----                                  RELATIVE  DEBUGGER      0013.000
11                          C-----                                  ADDRESS   SYMBOLS       0014.000
12                          C-----                                  --------  --------      0015.000
13                                                                                             0016.000
14                          C----- OBSERVE IN THE STORAGE DICTIONARY THE                       0017.000
15                          C----- RELATIVE ADDRESSES ASSIGNED FOR THE                        0018.000
16                          C----- VARIABLES AND CONSTANTS.                                   0019.000
17                                                                                             0020.000
18                          C----- OBSERVE IN THE SYMBOLIC CROSS REFERENCE                     0021.000
19                          C----- HOW THE OCCURRENCES OF THE VARIOUS                         0022.000
20                          C----- SYMBOLS ARE LOGGED.                                       0023.000
21                                                                                             0024.000
22                          C----- OBSERVE THE CORRELATION IN THE 'S.'                        0025.000
23                          C----- STATEMENT LABELS FOR THE SYMBOLIC                         0026.000
24                          C----- DEBUGGER USAGE BETWEEN THE SOURCE                         0027.000
25                          C----- LISTING AND THE OPTION 6 GENERATED                        0028.000
26                          C----- CODE LISTING.                                            0029.000
27                                                                                             0030.000
28                                INTEGER*8 REG67SAV                    ! P00008           0031.000
29                                INTEGER*8 IVAR1                       ! P00010           0032.000
30                                INTEGER*4 IVAR2                       ! P0001C           0033.000
31                                INTEGER*2 IVAR3                       ! P0001E           0034.000
32                                INTEGER*1 IVAR4                       ! P00020           0035.000
33                                CHARACTER*80 CVAR1                    ! P00020           0036.000
34                                CHARACTER*8  CVAR2                    ! P000D8           0037.000
35                                REAL*8 RVAR1                          ! P000E0           0038.000
36                                REAL*4 RVAR2                          ! P000F4           0039.000
37                                LOGICAL*4 LVAR1                       ! P000F0           0040.000
38                                LOGICAL*1 LVAR2                       ! P000FA           0041.000
39                                INTEGER*4 IADDRESS                    ! P000E8           0042.000
40                                                                                             0043.000
41                          C----- OBSERVE HOW THE EQUIVALENCE CAUSES THE                     0044.000
42                          C----- TWO VARIABLES TO HAVE THE SAME ADDRESS.                    0045.000
43                                EQUIVALENCE (IVAR4, CVAR1)            ! P00020           0046.000
44                                                                                             0047.000
45                          C----- OBSERVE HOW THE COMMON STATEMENT AFFECTS                    0048.000
46                          C----- THE RELATIVE ADDRESSES OF THE ITEMS                        0049.000
47                          C----- DEFINED TO BE IN COMMON.                                  0050.000
48                                INTEGER*8 CIVAR1                      ! C00000           0051.000
49                                INTEGER*4 CIVAR2                      ! C00008           0052.000
50                                INTEGER*2 CIVAR3                      ! C0000C           0053.000
51                                INTEGER*1 CIVAR4                      ! C0000E           0054.000
52                                                                                             0055.000
53                                COMMON / CEXAMPLE / CIVAR1,CIVAR2,                           0056.000
54                               &                    CIVAR3,CIVAR4     ! C00000           0057.000
```

840948

```
55                                                                                    0058.000
56                            DATA IVAR1, IVAR2 / 1, 2 /,                             0059.000
57                           *     IVAR3, IVAR4 / 3, 4 /                              0060.000
58                          C----- THE CONSTANT 1 GETS PUT INTO IVAR1        ! P00000  0061.000
59                          C----- THE CONSTANT 2 GETS PUT INTO IVAR2        ! P000EC  0062.000
60                          C----- THE CONSTANT 3 GETS PUT INTO IVAR3        ! P000F8  0063.000
61                          C----- THE CONSTANT 4 GETS PUT INTO IVAR4        ! P00000  0064.000
62                                                                                    0065.000
63   S.1                          CIVAR1 = IVAR1                             ! P000FC  S.1  0066.000
64   S.2                          CIVAR2 = IVAR2                             ! P00104  S.2  0067.000
65   S.3                          CIVAR3 = IVAR3                             ! P0010C  S.3  0068.000
66   S.4                          CIVAR4 = IVAR4                             ! P00114  S.4  0069.000
67                                                                                    0070.000
68   S.5                          RVAR1 = RVAR2 + 1.7                        ! P0011C  S.5  0071.000
69                          C----- THE CONSTANT 1.7 IS HEX 41D83333          ! P00230  0072.000
70                                                                                    0073.000
71                          C----- OBSERVE IN THE OPTION 6 GENERATED CODE              0074.000
72                          C----- LISTING HOW THE WRITE STATEMENT IS                 0075.000
73                          C----- BROKEN UP INTO VARIOUS CALLS TO THE                0076.000
74                          C----- SCIENTIFIC RUN-TIME LIBRARY.                      0077.000
75   S.6                          WRITE('LO',100,END=55,ERR=55) CIVAR1,               0078.000
76                           *       CIVAR2, CIVAR3, CIVAR4                   ! P0012C  S.6  0079.000
77                          C----- 'LO' IS REPRESENTED AS HEX 4C4F2020        ! P00234  0080.000
78                          C----- 100  REFERS TO LABEL )100                  ! P00050  0081.000
79                          C----- 55 REFERS TO LABEL )55                     ! P00224  S.26 0082.000
80                                                                                    0083.000
81                           100   FORMAT(1HO,'COMMON VARIABLE 1 = ', I20/             0084.000
82                           *        1H ,'COMMON VARIABLE 2 = ', I11/                 0085.000
83                           *        1H ,'COMMON VARIABLE 3 = ', I6/                  0086.000
84                           *        1H ,'COMMON VARIABLE 4 = ', I4)          ! P00050  0087.000
85                                                                                    0088.000
86   S.7                          CIVAR1=16Z7FFFFFFFFFFFFFFF                 ! P00160  S.7  0089.000
87                          C----- CONSTANT  HEX 7FFFFFFFFFFFFFFF            ! P00228  0090.000
88   S.8                          CIVAR2=817FFFFFFF                          ! P00168  S.8  0091.000
89                          C----- CONSTANT  HEX 7FFFFFFF                    ! P00238  0092.000
90   S.9                          CIVAR3=417FFF                              ! P00170  S.9  0093.000
91   S.10                         CIVAR4=217F                                ! P00178  S.10 0094.000
92                                                                                    0095.000
93   S.11                         WRITE('LO',100,END=55,ERR=55) CIVAR1,               0096.000
94                           *       CIVAR2, CIVAR3, CIVAR4                   ! P00180  S.11 0097.000
95                                                                                    0098.000
96                          C----- OBSERVE IN THE GENERATED CODE LISTING               0099.000
97                          C----- HOW THE INLINE CODE IS DISPLAYED.                 0100.000
98                                INLINE                                              0101.000
99   S.12                         STD  6,REG67SAV                            ! P00184  S.12 0102.000
100  S.13                         LD   6,IVAR1                               ! P00188  S.13 0103.000
101  S.14                         STD  6,CIVAR1                              ! P0018C  S.14 0104.000
102  S.15                         LW   7,IVAR2                               ! P001C0  S.15 0105.000
103  S.16                         STW  7,CIVAR2                              ! P001C4  S.16 0106.000
104  S.17                         LH   7,IVAR3                               ! P001C8  S.17 0107.000
105  S.18                         STH  7,CIVAR3                              ! P001CC  S.18 0108.000
106  S.19                         LB   7,IVAR4                               ! P001D0  S.19 0109.000
107  S.20                         STB  7,CIVAR4                              ! P001D4  S.20 0110.000
108  S.21                         LD   6,REG67SAV                            ! P001D8  S.21 0111.000
```

```
109                             ENDI                                                    0112.000
110                                                                                     0113.000
111                           C----- OBSERVE IN THE GENERATED CODE LISTING               0114.000
112                           C----- HOW THE LOGICAL VARIABLES ARE LOADED.              0115.000
113   S.22                          LVAR1 = .TRUE.                  ! P0010C   S.22     0116.000
114   S.23                          LVAR2 = .FALSE.                 ! P001E4   S.23     0117.000
115                                                                                     0118.000
116   S.24                          WRITE('LO',100,END=55,ERR=55) CIVAR1,               0119.000
117                               *       CIVAR2, CIVAR3, CIVAR4    ! P001E8   S.24     0120.000
118                                                                                     0121.000
119                           C----- OBSERVE IN THE GENERATED CODE LISTING               0122.000
120                           C----- HOW THE INLINE FUNCTION 'LOCF' IS                  0123.000
121                           C----- GENERATED.                                        0124.000
122   S.25                          IADDRESS = LOCF(IVAR1)          ! P0021C   S.25     0125.000
123                                                                                     0126.000
124   S.26                    55    STOP                            ! P00224   S.26     0127.000
125                                 END                             ! P00224   S.26     0128.000
```

B.2 Storage Dictionary

The storage dictionary facility provides a listing of symbolic names, statement numbers, and compiler-generated names contained within a program. Specifying $OPTION 3 or FORTRAN directive OPTION 3+ suppresses this listing.

B.2.1 Storage Dictionary Format

Under the heading of SYMBOL in the listing, the order is as follows:

1. Statement labels (each preceded by a right parenthesis) and compiler-generated symbols beginning with a right parenthesis.

2. Compiler-generated names for constants.

3. Symbolic names (user and compiler-generated) beginning with alphabetic characters.

Symbolic names are listed generally according to the ASCII collation sequence. Each listed item is identified under the following headings:

USAGE — Indicates the manner in which the specific item is used within a program; an item may be a constant, variable, array, label, common block, extended block, procedure, or namelist.

MODE — Indicates the type of an item, i.e., character, logical word, integer fullword. A statement number is listed as transfer or format depending upon its use.

STORAGE — Indicates whether the specific item is local or external to the program, or if it is an inactive, entry, or dummy item. For a common block, the size of the block is listed. If an item is within an extended block, the name of the extended block is listed.

LOCATION — Indicates the internal storage location of an item. The location is specified as a five-digit hexadecimal address containing one of the following prefixes:

- P signifies that the location is local to the program.
- X signifies that the location is external to the program.
- C signifies that the location is within a common block.
- I signifies that the location is an inactive item.
- \+ signifies that the location is an offset within an extended block.

An asterisk (*) before one of the prefixes indicates that the location contains the address of the item rather than the item itself.

A storage dictionary listing is provided for the program, EXAMPLE. This should be compared with the source listing in section B.1.

FORTRAN 77+ RELEASE 4.1.1 NATIVE MODE

SYMBOL	USAGE	MODE	STORAGE	LOCATION
)100	LABEL	FORMAT	LOCAL	P00050
)55	LABEL	TRANSFER	LOCAL	P00224
C.0001	CONSTANT	REAL	LOCAL	P00230
C.0002	CONSTANT	INTEGER FULLWORD	LOCAL	P00234
C.0003	CONSTANT	INTEGER FULLWORD	LOCAL	P00238
C.0004	CONSTANT	INTEGER DOUBLE	LOCAL	P00228
A.TF	PROCEDURE	REAL	EXTERNAL	X00200
CEXAMPLE	COMMON BLOCK		S00000F	
CIVAR1	VARIABLE	INTEGER DOUBLE	CEXAMPLE	C00000
CIVAR2	VARIABLE	INTEGER FULLWORD	CEXAMPLE	C00008
CIVAR3	VARIABLE	INTEGER HALFWORD	CEXAMPLE	C0000C
CIVAR4	VARIABLE	INTEGER BYTE	CEXAMPLE	C0000E
CVAR1	VARIABLE	CHARACTER*80	LOCAL	P00000
CVAR2	VARIABLE	CHARACTER*8	LOCAL	P000C8
E.XI	PROCEDURE	REAL	EXTERNAL	X00224
EXAMPLE	PROCEDURE	REAL	ENTRY	P000FC
IADDRESS	VARIABLE	INTEGER FULLWORD	LOCAL	P000E8
IVAR1	VARIABLE	INTEGER DOUBLE	LOCAL	P000D0
IVAR2	VARIABLE	INTEGER FULLWORD	LOCAL	P000EC
IVAR3	VARIABLE	INTEGER HALFWORD	LOCAL	P000F8
IVAR4	VARIABLE	INTEGER BYTE	LOCAL	P00000
LVAR1	VARIABLE	LOGICAL WORD	LOCAL	P000F0
LVAR2	VARIABLE	LOGICAL BYTE	LOCAL	P000FA
REG67SAV	VARIABLE	INTEGER DOUBLE	LOCAL	P000D8
RVAR1	VARIABLE	DOUBLE	LOCAL	P000E0
RVAR2	VARIABLE	REAL	LOCAL	P000F4
T.EF	PROCEDURE	REAL	EXTERNAL	X00218
W.IF	PROCEDURE	REAL	EXTERNAL	X001E8

B.3 Symbolic Cross-Reference

The symbolic cross-reference facility provides all the symbolic names and statement numbers within a program, listed side-by-side with all the source line numbers at which they are referenced. To suppress this listing specify $OPTION 4 in job control or specify the OPTION 4+ directive.

B.3.1 Symbolic Cross-Reference Format

Symbolic names are listed in alphabetic order. Statement numbers are listed in sequential order, sorted on the leading (i.e., leftmost) significant digit. For example, statement number 100 precedes statement number 20, which precedes statement number 3. Statement numbers with a common leading digit are listed in descending numerical sequence. Nonsignificant zeros and spaces are suppressed. For example, references to 00600, 600, and 6Ø0Ø0 will all be listed under symbol 600.

The symbolic name and statement number items that are included in text strings or in comments are not listed in cross-reference output.

Multiple occurrences of a symbolic or statement number within a single source line are all listed. Continuation source statement lines are identified separately from the initial source statement line.

Symbolic names occurring in EQUIVALENCE statements are identified by the code EQV instead of a source line number for each occurrence.

A symbolic cross-reference listing is provided for the program, EXAMPLE. This should be compared with the source listing in section B.1.

```
FORTRAN 77+ RELEASE 4.1.1   NATIVE MODE
CROSS REFERENCE FOR EXAMPLE
          SYMBOL      SOURCE LINE


           100        0075    0081    0093    0116
           55         0075    0075    0093    0093    0116    0116    0124
          CEXAMPLE    0053
          CIVAR1      0048    0053    0063    0075    0085    0093    0101    0116
          CIVAR2      0049    0053    0064    0075    0088    0093    0103    0116
          CIVAR3      0050    0053    0065    0076    0090    0094    0105    0117
          CIVAR4      0051    0054    0066    0076    0091    0094    0107    0117
          CVAR1       0033    0043
          CVAR2       0034
          EXAMPLE     0001
          IADDRESS    0039    0122
          IVAR1       0029    0056    0063    0100    0122
          IVAR2       0030    0056    0064    0102
          IVAR3       0031    0056    0065    0104
          IVAR4       0032    0043    0057    0066    0106
          LOCF        0122
          LVAR1       0037    0113
          LVAR2       0038    0114
          REG67SAV    0028    0099    0108
          RVAR1       0035    0068
          RVAR2       0036    0068
```

B.4 Generated Code Listings

The generated code listing facility provides a listing of relative addresses, hexadecimal machine code instructions and data area contents, statement labels and compiler generated statement markers for use with the Symbolic Debugger, and instruction equivalent assembler code. Specifying $OPTION 6 or FORTRAN 77+ directive OPTION 6+ enables the output of this listing. The absence of $OPTION 6 or the presence of the FORTRAN 77+ directive OPTION 6- suppresses this listing.

B.4.1 Generated Code Listing Format

The generated code listing begins with the name of the source module, whether it is a program, subroutine, function, etc., in the upper left hand corner of the listing. If listed output (LO) is assigned to system listed output (SLO), the module name appears in the upper left hand corner of every page in the listing.

The listing appears below the name. The format of each line of the page is as follows:

- A relative memory address is slightly indented from the left margin.
- The hexadecimal contents of the memory address appear to the right of the address. This may be machine coded instructions or data.
- The statement labels or the compiler generated statement markers for use with the Symbolic Debugger under MPX-32 appear to the right of the memory contents. Statement labels have the form n where n is the statement number contained in the input source record. The statement markers have the form S.m where m is a compiler assigned statement number. Not all lines in the generated code listing have statement labels or markers. Data areas such as those generated from FORMAT statements and executable instructions may contain statement labels. Statement markers appear only on the first executable instruction for a set of instructions which were generated from a FORTRAN 77+ source statement.
- The instruction equivalent assembler code appears to the right of the statement label and marker field. The field is made up of the assembler instruction mnemonic followed by the operand for the particular instruction. For example, LD 6,IVAR1.

This format is maintained throughout the compilation of the entire program unit unless the source listing is also selected. In that case, the two listings appear to be interspersed.

A generated code listing is provided for the program, EXAMPLE. This should be compared with the source listing in section B.1.

```
00000  00000000
00004  00000001
000EC  00000002
000F8  0003
00000  04
000FC                  S.1     EQU     $
00050  8F110130        )100
00054  1E434F4D
00058  4D4F4E20
0005C  56415249
00060  41424C45
00064  20312030
00068  208E0114
0006C  00131101
00070  201E434F
00074  4D4D4F4E
00078  20564152
0007C  4941424C
00080  45203220
00084  30208E01
00088  08001311
0008C  01201E43
00090  4F4D4D4F
00094  4E205641
00098  52494142
0009C  4C452033
000A0  2030208E
000A4  01060013
000A8  1101201E
000AC  434F4D4D
000B0  4F4E2056
000B4  41524941
000B8  424C4520
000BC  34203020
000C0  8E010400
000C4  08
000FC  AF000002                LD      6,IVAR1
00100  D7000002                STD     6,CIVAR1
00104                  S.2     EQU     $
00104  AF8000EC                LW      7,IVAR2
00108  D7800008                STW     7,CIVAR2
0010C                  S.3     EQU     $
0010C  AF8000F9                LH      7,IVAR3
00110  D7800000                STH     7,CIVAR3
00114                  S.4     EQU     $
00114  AF880000                LB      7,IVAR4
00118  D788000E                STB     7,CIVAR4
0011C                  S.5     EQU     $
0011C  AF8000F4                LW      7,RVAR2
00120  E3880230                ADFW    7,C.0001
00124  2F70                    TRR     7,6
00126      0FF0                IR      7
00128  D700002                 STD     6,RVAR1
0012C                  S.6     EQU     $
```

B10953

```
00012C  F8800001          BL      W.IF
00130   00000010
00134   02000234          WORD    C.0002
00138   00080050          WORD    )100
0013C   01000225          WORD    )55
00140   01000225          WORD    )55
00144   F8800001          BL      A.IF
00148   00000010
0014C   03000002          WORD    CIVAR1
00150   02000008          WORD    CIVAR2
00154   0100000D          WORD    CIVAR3
00158   0008000E          WORD    CIVAR4
0015C   F8800001          BL      T.EF
00160           S.7       EQU     $
00160   AF000224          LD      6,C.0004
00164   D7000002          STD     6,CIVAR1
00168           S.8       EQU     $
00168   AF800238          LW      7,C.0003
0016C   D7800008          STW     7,CIVAR2
00170           S.9       EQU     $
00170   C8807FFF          LI      7,32767
00174   D780000D          STH     7,CIVAR3
00178           S.10      EQU     $
00178   C880007F          LI      7,127
0017C   D788000E          STB     7,CIVAR4
00180           S.11      EQU     $
00180   F880012D          BL      W.IF
00184   00000010
00188   02000234          WORD    C.0002
0018C   00080050          WORD    )100
00190   01000225          WORD    )55
00194   01000225          WORD    )55
00198   F8800145          BL      A.IF
0019C   00000010
001A0   03000002          WORD    CIVAR1
001A4   02000008          WORD    CIVAR2
001A8   0100000D          WORD    CIVAR3
001AC   0008000E          WORD    CIVAR4
001B0   F880015D          BL      T.EF
00184           S.12      EQU     $
00184   D700000A          STD     6,REG67SAV
00188           S.13      EQU     $
00188   AF000002          LD      6,IVAR1
0018C           S.14      EQU     $
0018C   D7000002          STD     6,CIVAR1
001C0           S.15      EQU     $
001C0   AF8000EC          LW      7,IVAR2
001C4           S.16      EQU     $
001C4   D7800008          STW     7,CIVAR2
001C8           S.17      EQU     $
001C8   AF8000F9          LH      7,IVAR3
001CC           S.18      EQU     $
001CC   D780000D          STH     7,CIVAR3
001D0           S.19      EQU     $
```

EXAMPLE

```
001D0   AF880000            LB      7,IVAR4
001D4           S.20        EQU     $
001D4   D788000E            STB     7,CIVAR4
001D8           S.21        EQU     $
001D8   AF0000DA            LD      6,REG67SAV
001DC           S.22        EQU     $
001DC   C88000FF            LI      7,255
001E0   D78000F0            STW     7,LVAR1
001E4           S.23        EQU     $
001E4   F80800FA            IMB     LVAR2
001E8           S.24        EQU     $
001E8   F8800181            BL      W.IF
001EC   00000010
001F0   02000234            WORD    C.0002
001F4   00080050            WORD    )100
001F8   01000225            WORD    )55
001FC   01000225            WORD    )55
00200   F8800199            BL      A.TF
00204   00000010
00208   03000002            WORD    CIVAR1
0020C   02000008            WORD    CIVAR2
00210   01000000            WORD    CIVAR3
00214   0008000E            WORD    CIVAR4
00218   F8800181            BL      T.EF
0021C           S.25        EQU     $
0021C   37800002            LA      7,IVAR1
00220   D78000E8            STW     7,IADDRESS
00224           S.26        EQU     $
00224   F8800001    )55     BL      E.XI
00228   7FFFFFFF
0022C   FFFFFFFF
00230   41183333
00234   4C4F2020
00238   7FFFFFFF
```

APPENDIX C

ASCII CODE SET

HEX	DEC	ASCII GRAPHIC	CARD CODE (IBM029)	INTERNAL BINARY	ASCII NAME
00	0		12-0-9-8-1	0000 0000	NUL
01	1		12-9-1	0000 0001	SOH
02	2		12-9-2	0000 0010	STX
03	3		12-9-3	0000 0011	ETX
04	4		9-7	0000 0100	EOT
05	5		0-9-8-5	0000 0101	ENQ
06	6		0-9-8-6	0000 0110	ACK
07	7		0-9-8-7	0000 0111	BEL
08	8		11-9-6	0000 1000	BS
09	9		12-9-5	0000 1001	HT
0A	10		0-9-5	0000 1010	LF
0B	11		12-9-8-3	0000 1011	VT
0C	12		12-9-8-4	0000 1100	FF
0D	13		12-9-8-5	0000 1101	CR
0E	14		12-9-8-6	0000 1110	SO
0F	15		12-9-8-7	0000 1111	SI
10	16		12-11-9-8-1	0001 0000	DLE
11	17		11-9-1	0001 0001	C1
12	18		11-9-2	0001 0010	DC2
13	19		11-9-3	0001 0011	DC3
14	20		9-8-4	0001 0100	DC4
15	21		9-8-5	0001 0101	NAK
16	22		9-2	0001 0110	SYN
17	23		0-9-6	0001 0111	ETB
18	24		11-9-8	0001 1000	CAN
19	25		11-9-8-1	0001 1001	EM
1A	26		9-8-7	0001 1010	SUB
1B	27		0-9-7	0001 1011	ESC
1C	28		11-9-8-4	0001 1100	FS
1D	29		11-9-8-5	0001 1101	GS
1E	30		11-9-8-6	0001 1110	RS
1F	31		11-9-8-7	0001 1111	US

HEX	DEC	ASCII GRAPHIC	CARD CODE (IBM029)	INTERNAL BINARY	ASCII NAME
20	32			0010 0000	SP
21	33	!	12-8-7	0010 0001	Exclamation Point
22	34	"	8-7	0010 0010	Quotation Marks
23	35	#	8-3	0010 0011	Number Sign
24	36	$	11-8-3	0010 0100	Dollar Sign
25	37	%	0-8-4	0010 0101	Percent
26	38	&	12	0010 0110	Ampersand
27	39	'	8-5	0010 0111	Apostrophe
28	40	(	12-8-5	0010 1000	Opening Parenthesis
29	41	)	11-8-5	0010 1001	Closing Parenthesis
2A	42	*	11-8-4	0010 1010	Asterisk
2B	43	+	12-8-6	0010 1011	Plus
2C	44	,	0-8-3	0010 1100	Comma
2D	45	-	11	0010 1101	Hyphen
2E	46	.	12-8-3	0010 1110	Period
2F	47	/	0-1	0010 1111	Slant
30	48	0	0	0011 0000	Zero
31	49	1	1	0011 0001	One
32	50	2	2	0011 0010	Two
33	51	3	3	0011 0011	Three
34	52	4	4	0011 0100	Four
35	53	5	5	0011 0101	Five
36	54	6	6	0011 0110	Six
37	55	7	7	0011 0111	Seven
38	56	8	8	0011 1000	Eight
39	57	9	9	0011 1001	Nine
3A	58	:	8-2	0011 1010	Colon
3B	59	;	11-8-6	0011 1011	Semicolon
3C	60	<	12-8-4	0011 1100	Less Than
3D	61	=	8-6	0011 1101	Equals
3E	62	>	0-8-6	0011 1110	Greater Than
3F	63	?	0-8-7	0011 1111	Question Mark
40	64	@	8-4	0100 0000	Commercial At
41	65	A	12-1	0100 0001	Uppercase A
42	66	B	12-2	0100 0010	Uppercase B
43	67	C	12-3	0100 0011	Uppercase C
44	68	D	12-4	0100 0100	Uppercase D
45	69	E	12-5	0100 0101	Uppercase E
46	70	F	12-6	0100 0110	Uppercase F
47	71	G	12-7	0100 0111	Uppercase G

HEX	DEC	ASCII GRAPHIC	CARD CODE (IBM029)	INTERNAL BINARY	ASCII NAME
48	72	H	12-8	0100 1000	Uppercase H
49	73	I	12-9	0100 1001	Uppercase I
4A	74	J	11-1	0100 1010	Uppercase J
4B	75	K	11-2	0100 1011	Uppercase K
4C	76	L	11-3	0100 1100	Uppercase L
4D	77	M	11-4	0100 1101	Uppercase M
4E	78	N	11-5	0100 1110	Uppercase N
4F	79	O	11-6	0100 1111	Uppercase O
50	80	P	11-7	0101 0000	Uppercase P
51	81	Q	11-8	0101 0001	Uppercase Q
52	82	R	11-9	0101 0010	Uppercase R
53	83	S	0-2	0101 0011	Uppercase S
54	84	T	0-3	0101 0100	Uppercase T
55	85	U	0-4	0101 0101	Uppercase U
56	86	V	0-5	0101 0110	Uppercase V
57	87	W	0-6	0101 0111	Uppercase W
58	88	X	0-7	0101 1000	Uppercase X
59	89	Y	0-8	0101 1001	Uppercase Y
5A	90	Z	0-9	0101 1010	Uppercase Z
5B	91	[	12-8-2	0101 1011	Opening Bracket
5C	92	\	0-8-2	0101 1100	Reverse Slant
5D	93	]	11-8-2	0101 1101	Closing Bracket
5E	94	^	11-8-7	0101 1110	Circumflex
5F	95	_	0-8-5	0101 1111	Underline
60	96	`	8-1	0110 0000	Accent Grave
61	97	a	12-0-1	0110 0001	Lowercase a
62	98	b	12-0-2	0110 0010	Lowercase b
63	99	c	12-0-3	0110 0011	Lowercase c
64	100	d	12-0-4	0110 0100	Lowercase d
65	101	e	12-0-5	0110 0101	Lowercase e
66	102	f	12-0-6	0110 0110	Lowercase f
67	103	g	12-0-7	0110 0111	Lowercase g
68	104	h	12-0-8	0110 1000	Lowercase h
69	105	i	12-0-9	0110 1001	Lowercase i
6A	106	j	12-11-1	0110 1010	Lowercase j
6B	107	k	12-11-2	0110 1011	Lowercase k
6C	108	l	12-11-3	0110 1100	Lowercase l
6D	109	m	12-11-4	0110 1101	Lowercase m
6E	110	n	12-11-5	0110 1110	Lowercase n
6F	111	o	12-11-6	0110 1111	Lowercase o

HEX	DEC	ASCII GRAPHIC	CARD CODE (IBM029)	INTERNAL BINARY	ASCII NAME
70	112	p	12-11-7	0111 0000	Lowercase p
71	113	q	12-11-8	0111 0001	Lowercase q
72	114	r	12-11-9	0111 0010	Lowercase r
73	115	s	11-0-2	0111 0011	Lowercase s
74	116	t	11-0-3	0111 0100	Lowercase t
75	117	u	11-0-4	0111 0101	Lowercase u
76	118	v	11-0-5	0111 0110	Lowercase v
77	119	w	11-0-6	0111 0111	Lowercase w
78	120	x	11-0-7	0111 1000	Lowercase x
79	121	y	11-0-8	0111 1001	Lowercase y
7A	122	z	11-0-9	0111 1010	Lowercase z
7B	123	{	12-0	0111 1011	Opening Brace
7C	124	|	12-11	0111 1100	Vertical Line
7D	125	}	11-0	0111 1101	Closing Brace
7E	126	~	11-0-1	0111 1110	Tilde
7F	127		12-9-7	0111 1111	DEL
80	128		11-0-9-8-1	1000 0000	
81	129		0-9-1	1000 0001	
82	130		0-9-2	1000 0010	
83	131		0-9-3	1000 0011	
84	132		0-9-4	1000 0100	
85	133		11-9-5	1000 0101	
86	134		12-9-6	1000 0110	
87	135		11-9-7	1000 0111	
88	136		0-9-8	1000 1000	
89	137		0-9-8-1	1000 1001	
8A	138		0-9-8-2	1000 1010	
8B	139		0-9-8-3	1000 1011	
8C	140		0-9-8-4	1000 1100	
8D	141		12-9-8-1	1000 1101	
8E	142		12-9-8-2	1000 1110	
8F	143		11-9-8-3	1000 1111	
90	144		12-11-0-9-8-1	1001 0000	
91	145		9-1	1001 0001	
92	146		11-9-8-2	1001 0010	
93	147		9-3	1001 0011	
94	148		9-4	1001 0100	
95	149		9-5	1001 0101	
96	150		9-6	1001 0110	
97	151		12-9-8	1001 0111	

HEX	DEC	ASCII GRAPHIC	CARD CODE (IBM029)	INTERNAL BINARY	ASCII NAME
98	152		9-8	1001 1000	
99	153		9-8-1	1001 1001	
9A	154		9-8-2	1001 1010	
9B	155		9-8-3	1001 1011	
9C	156		12-9-4	1001 1100	
9D	157		11-9-4	1001 1101	
9E	158		9-8-6	1001 1110	
9F	159		11-0-9-1	1001 1111	
A0	160		12-0-9-1	1010 0000	
A1	161		12-0-9-2	1010 0001	
A2	162		12-0-9-3	1010 0010	
A3	163		12-0-9-4	1010 0011	
A4	164		12-0-9-5	1010 0100	
A5	165		12-0-9-6	1010 0101	
A6	166		12-0-9-7	1010 0110	
A7	167		12-0-9-8	1010 0111	
A8	168		12-8-1	1010 1000	
A9	169		12-11-9-1	1010 1001	
AA	170		12-11-9-2	1010 1010	
AB	171		12-11-9-3	1010 1011	
AC	172		12-11-9-4	1010 1100	
AD	173		12-11-9-5	1010 1101	
AE	174		12-11-9-6	1010 1110	
AF	175		12-11-9-7	1010 1111	
B0	176		12-11-9-8	1011 0000	
B1	177		11-8-1	1011 0001	
B2	178		11-0-9-2	1011 0010	
B3	179		11-0-9-3	1011 0011	
B4	180		11-0-9-4	1011 0100	
B5	181		11-0-9-5	1011 0101	
B6	182		11-0-9-6	1011 0110	
B7	183		11-0-9-7	1011 0111	
B8	184		11-0-9-8	1011 1000	
B9	185		0-8-1	1011 1001	
BA	186		12-11-0	1011 1010	
BB	187		12-11-0-9-1	1011 1011	
BC	188		12-11-0-9-2	1011 1100	
BD	189		12-11-0-9-3	1011 1101	
BE	190		12-11-0-9-4	1011 1110	
BF	191		12-11-0-9-5	1011 1111	

HEX	DEC	ASCII GRAPHIC	CARD CODE (IBM029)	INTERNAL BINARY	ASCII NAME
C0	192		12-11-0-9-6	1100 0000	
C1	193		12-11-0-9-7	1100 0001	
C2	194		12-11-0-9-8	1100 0010	
C3	195		12-0-8-1	1100 0011	
C4	196		12-0-8-2	1100 0100	
C5	197		12-0-8-3	1100 0101	
C6	198		12-0-8-4	1100 0110	
C7	199		12-0-8-5	1100 0111	
C8	200		12-0-8-6	1100 1000	
C9	201		12-0-8-7	1100 1001	
CA	202		12-11-8-1	1100 1010	
CB	203		12-11-8-2	1100 1011	
CC	204		12-11-8-3	1100 1100	
CD	205		12-11-8-4	1100 1101	
CE	206		12-11-8-5	1100 1110	
CF	207		12-11-8-6	1100 1111	
D0	208		12-11-8-7	1101 0000	
D1	209		11-0-8-1	1101 0001	
D2	210		11-0-8-2	1101 0010	
D3	211		11-0-8-3	1101 0011	
D4	212		11-0-8-4	1101 0100	
D5	213		11-0-8-5	1101 0101	
D6	214		11-0-8-6	1101 0110	
D7	215		11-0-8-7	1101 0111	
D8	216		12-11-0-8-1	1101 1000	
D9	217		12-11-0-1	1101 1001	
DA	218		12-11-0-2	1101 1010	
DB	219		12-11-0-3	1101 1011	
DC	220		12-11-0-4	1101 1100	
DD	221		12-11-0-5	1101 1101	
DE	222		12-11-0-6	1101 1110	
DF	223		12-11-0-7	1101 1111	
E0	224		12-11-0-8	1110 0000	
E1	225		12-11-0-9	1110 0001	
E2	226		12-11-0-8-2	1110 0010	
E3	227		12-11-0-8-3	1110 0011	
E4	228		12-11-0-8-4	1110 0100	
E5	229		12-11-0-8-5	1110 0101	
E6	230		12-11-0-8-6	1110 0110	
E7	231		12-11-0-8-7	1110 0111	

HEX	DEC	ASCII GRAPHIC	CARD CODE (IBM029)	INTERNAL BINARY	ASCII NAME
E8	232		12-0-9-8-2	1110 1000	
E9	233		12-0-9-8-3	1110 1001	
EA	234		12-0-9-8-4	1110 1010	
EB	235		12-0-9-8-5	1110 1011	
EC	236		12-0-9-8-6	1110 1100	
ED	237		12-0-9-8-7	1110 1101	
EE	238		12-11-9-8-2	1110 1110	
EF	239		12-11-9-8-3	1110 1111	
F0	240		12-11-9-8-4	1111 0000	
F1	241		12-11-9-8-5	1111 0001	
F2	242		12-11-9-8-6	1111 0010	
F3	243		12-11-9-8-7	1111 0011	
F4	244		11-0-9-8-2	1111 0100	
F5	245		11-0-9-8-3	1111 0101	
F6	246		11-0-9-8-4	1111 0110	
F7	247		11-0-9-8-5	1111 0111	
F8	248		11-0-9-8-6	1111 1000	
F9	249		11-0-9-8-7	1111 1001	
FA	250		12-11-0-9-8-2	1111 1010	
FB	251		12-11-0-9-8-3	1111 1011	
FC	252		12-11-0-9-8-4	1111 1100	
FD	253		12-11-0-9-8-5	1111 1101	
FE	254		12-11-0-9-8-6	1111 1110	
FF	255		12-11-0-9-8-7	1111 1111	

APPENDIX D

DIAGNOSTICS

D.1 Compile-Time Diagnostics

The compiler generates error messages when rules of the FORTRAN 77+ language have been violated. These violations, in areas such as syntax, parameter usage, storage requirements, and variable typing, are reported as either source line errors or context errors.

When the compiler's interaction with the operating system causes abnormal termination of compilation, "FTxx" abort codes are generated. Source line errors, context errors, and abort codes are described in the sections that follow.

D.1.1 Source Line Errors

Errors detected by the compiler in FORTRAN 77+ source input statements are reported with program-listed output messages that generally precede the source line affected.

The descriptive message qualifies the error as to:

- Terminal (T) or warning (W) in nature.
- Source program line number and column number where the error was detected for errors in source syntax.

A brief description of the error type and the last eight nonblank characters scanned before error detection are listed as part of the error message or, in some cases, a relevant symbolic name is listed.

Warning (W) errors result in a continuation of statement scan by the compiler and an attempt to compile the statement.

Terminal (T) errors result in termination of statement scan for the source program statement in which the error was detected.

For certain errors encountered in EQUIVALENCE statements, line numbers are identified in the listed output message. Generally, however, the word EQUIVALENCE, followed by the symbol involved in the error, is specified.

D.1.2 Context Errors

Syntactically correct statements that result in context errors because of interaction with other program elements are detected during the code generation phase of compilation. The listed output message for these types of errors specifies relative object code location rather than source statement line and column number. The erroneous symbol involved is also specified.

The total number of error messages issued, if any, is listed when compilation is complete. This error summary has the following format:

```
*ERRORS nn
```

nn The total number of terminal and warning errors detected during program compilation.

The error OPTIMIZER LOST POINTER is an internal compiler error resulting from loss of essential register contents during the code optimization phase. If this error occurs, submit a Software Problem Report (SPR) to Gould CSD along with a copy of the code that produced the error.

A sample listed output with error messages follows:

```
GOULD CSD FORTRAN 77+ (84 APR 02/ RELEASE 4.1)

MAIN

*** W ERR FOUND AT LINE 00001          COL 12 DIMENT      STATEMENT MISSPELLING
1                                      DIMENTION A(2)
2                                      COMMON B,C
3                                      EQUIVALENCE (B,C)
*** W ERR FOUND AT LINE 00004          COL 14 FORMAT (    STATEMENT NUMBER
                                                          MISSING
*** W ERR FOUND AT LINE 00004          COL 27 UMBER',N    ILLEGAL CHARACTER FOR
                                                          SYNTAX
*** W ERR FOUND AT LINE 00004          COL 28 MBER',N)    ILLEGAL CHARACTER FOR
                                                          SYNTAX
4                                      FORMAT ('NO NUMBER',N)
*** W ERR FOUND IN EQUIVALENCE    C                       ATTEMPT TO EQUIVALENCE
                                                          TO PREVIOUSLY USED
                                                          COMMON ITEMS
5                                      DATA A/1/
6                                      DATA B/1.0/
***  W  ERR  FOUND  AT  LOC.  00000 B                     ATTEMPT TO INITIALIZE
                                                          BLANK COMMON
7                                      DO 99 I=1,10
8                                      GO TO 100
*** W ERR FOUND AT LINE 00009 COL 10 STOP                 NO PATH TO THIS
                                                          STATEMENT
9                                      STOP
*** W ERR FOUND AT LINE 00010          )99                MISSING DO LOOP TERMINAL
                                                          LABEL
*** W ERR FOUND AT LINE 00010 COL 72 END                  ALLOCATION OVERFLOW
*** W ERR FOUND AT LOC. 00030          )100               UNDEFINED SYMBOL
10                                     END
*ERRORS IN MAIN                        10
```

D.1.3 Abort Codes - FT

CODE	DESCRIPTION
FT01	PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE FORTRAN SCRATCH FILE *U1. If logical file code *U1 is assigned to temporary disc space, the maximum number of extends was reached. Specify a larger size TEMP file to reduce the number of extends needed.
FT02	PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE FORTRAN SCRATCH FILE *U2. If the logical file code *U2 is assigned to temporary disc space, the maximum number of extends was reached. Specify a larger size TEMP file to reduce the number of extends needed.
FT03	PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE FORTRAN BINARY OUTPUT (BO) FILE. If logical file code BO is assigned to the system binary output (SBO), then the maximum number of extends were reached. SBO can be increased by specifying the SBO=size on the $JOB directive. If logical file code BO is assigned to a permanent disc file, the file was too small and could not be extended, or the maximum number of extends was reached. Recreate the file with a larger size so it can be extended.
FT04	PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE FORTRAN BINARY OUTPUT (GO) FILE. If logical file code GO is assigned to the system generated output (SGO), then the maximum number of extends were reached. SGO can be increased by specifying the SGO=size on the $JOB directive. If logical file code GO is assigned to a permanent disc file, the file was too small and could not be extended, or the maximum number of extends was reached. Recreate the file with a larger size so it can be extended.
FT05	PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE FORTRAN LISTED OUTPUT (LO). If logical file code LO is assigned to the system listed output (SLO), then the maximum number of extends were reached. SLO can be increased by specifying the SLO=size on the $JOB directive. If logical file code LO is assigned to a permanent disc file, the file was too small and could not be extended, or the maximum number of extends was reached. Recreate the file with a larger size so it can be extended.

CODE	DESCRIPTION
FT06	COMPILATION ERRORS WERE DETECTED. ERRORS DESCRIBED IN LISTED OUTPUT (LO). Source line and/or context errors were detected during compilation. Their description and location in the source code is described in the output from logical file code LO. This abort code is available for conditional job termination.
FT07	REALLOCATION OF INCLUDE FILE WAS DENIED BY M.ASSN. In attempting to reallocate an INCLUDE file from within a nested structure, the file was no longer available for allocation. This could indicate that the file is in use, has been deleted, or the access rights have changed since the last time it was used by the compiler.
FT08	COMPILER STACK AREA OVERFLOW. PROGRAM TOO LARGE TO COMPILE. The 32K area in the compiler for the symbol table and compiler stacks has been exceeded. The program source module must be divided into smaller compilation units (subroutines, functions, etc.) to be compiled.
FT09	COMPILER FAULT. TRYING TO ALLOCATE EXCESSIVE MAPS. An internal logic error in the compiler has caused it to try to allocate an excessive number of memory maps using the M.GD service. Please submit a Software Problem Report if this occurs.
FT10	COMPILER ALLOCATION EXCEEDS PHYSICAL MEMORY. This is most likely an internal logic error in the compiler using the M.GD service, but it could be caused if at least 256KW of memory was not sysgened into the system.
FT11	COMPILER FAULT. M.MEMB SERVICE IN USE. An internal logic error in the compiler has caused a conflict between the M.GD service and the M.MEMB service. Please submit a Software Problem Report if this occurs.
FT12	M.OPENR could not be performed for LFC (BO). This indicates that the file assigned to LFC (BO) was in use at the time the compiler attempted to open it.
FT13	M.OPENR could not be performed for LFC (GO). This indicates that the file assigned to LFC (GO) was in use at the time the compiler attempted to open it.

FT14 M.OPENR could not be performed for LFC (LO).

This indicates that the file assigned to LFC (LO) was in use at the time the compiler attempted to open it.

FT15 M.OPENR could not be performed for LFC (SI).

This indicates that the file assigned to LFC (SI) was in use at the time the compiler attempted to open it.

FT16 M.OPENR could not be performed for cross reference.

This indicates that the file assigned for cross reference was in use at the time the compiler attempted to open it.

D.2 Execution-Time Diagnostics

While running FORTRAN 77+ compiled user task (not to be confused with the actual compilation of that task) or while running a task that uses the Scientific Run-Time Library (SRTL), several potential error indications can be encountered.

If the IOSTAT parameter is used in the task's I/O statements, an integer value is generated when I/O is performed. These values are described below.

When a task is forced by the operating system or the SRTL to terminate abnormally, "RTxx" and "RSxx" abort codes are generated. The RT prefix may be replaced by a W. or T. (indicating whether the error was warning or terminal, respectively). The abort codes are described in the following sections.

D.2.1 IOSTAT Values

For all input/output statements, if an input/output status specifier (IOSTAT=ios) is present, the status for the operation will be returned unless abnormal termination of the program results during execution of the operation. The input/output status will be returned as an integer word to the user-specified location. The value returned indicates status for the operation (with the exception of the OPEN, CLOSE, and INQUIRE operations) as follows:

Integer Value	Status
ios = 0	No error encountered, operation successfully completed.
ios < 0	EOF/EOM encountered during I/O operation. See status bits below.
0 < ios < 100	Integer value corresponds to error code RTxx where IOS=xx.
ios $\geq$ 100	IOCS error detected. See FCB status bits below.

Status Bits From the FCB Status Word:

Bit	Meaning
0	Set if EOF/EOM encountered (forces IOS to be negative).
1	Error condition found.
2	Invalid blocking buffer control pointers have been encountered during file blocking or deblocking.
3	Write protect violation.
4	Device inoperable.
5	Beginning-of-medium (BOM) (load point) or illegal volume number (multivolume magnetic tape).
6	End-of-file.
7	End-of-medium (end-of-tape, end-of-disc file).
8-11	Specifies general testing status received from an 8000 level test device instruction.
12-15	Specified DCC testing status received from a 4000 level test device instruction.
16-31	Specifies a device status received from a 2000 level test device instruction. These bits are not applicable for the X-Y plotter, paper tape, card reader, and teletypewriter. Bit meaning for a particular device can be found in MPX-32 Reference Manual.

The following IOSTAT values are returned only with the OPEN, CLOSE, and INQUIRE statements.

Native Mode

Integer Value	Status
0	No errors, operation complete.
1	A status of OLD was specified for a nonexistent file.
2	A status of NEW was specified for an existing file.
3	A status of OLD/NEW was specified but no FILE was given.
4	Invalid start address; negative value illegal.
5	Alternate unit is not connected.
6	Spoolfile name and queue name both specified but do not match.
7	Invalid DENSITY specification.
8	Invalid access restriction.
9	DENSITY was specified but DEVICE was not.
10	Multiple assignment type (i.e., only one of FILE, DEVICE, ALTUNIT may appear at one time).
11	Invalid CONTROLBITS specification.
12	UNIT specifier missing.
13	No FILE, DEVICE, or ALTUNIT was given.
14	Invalid ACCESS specification.
15	VOLUME was specified for a nontape device.
16	An argument other than UNIT was specified with ALTUNIT.
17	Invalid BLANK specification.
18	Invalid FORM specification.
19	RECL value too large.
20	RECL specifier missing for direct access.
21	RECL value must be positive.
22	Invalid STATUS specification.
23	Invalid QUEUE specification.
24	Disc devices may not be accessed.
26	Pathname invalid.
27	Pathname consists of volume only.
28	Volume not mounted.
29	Directory does not exist.
30	Directory name in use.
31	Directory creation not allowed at specified level.
32	Resource does not exist.

Native Mode

Integer Value	Status
33	Resource name in use.
34	Resource descriptor unavailable.
35	Directory entry unavailable.
36	Required file space unavailable.
37	Unrecoverable I/O error while reading DMAP.
38	Unrecoverable I/O error while writing DMAP.
39	Unrecoverable I/O error while reading resource descriptor.
40	Unrecoverable I/O error while writing resource descriptor.
41	Unrecoverable I/O error while reading SMAP.
42	Unrecoverable I/O error while writing SMAP.
43	Unrecoverable I/O error while reading directory.
44	Unrecoverable I/O error while writing directory.
45	Projectgroup name invalid.
46	Projectgroup key invalid.
47	FCB destroyed.
48	Parameter address error.
49	Resource descriptor not currently allocated.
50	Pathname block overflow.
51	File space not currently allocated.
52	'Change defaults' not allowed.
53	Cannot access resource in required mode.
54	Operation not allowed on this resource type.
55	Required parameter was not specified.
56	File extension denied; segment definition area full.
57	File extension denied, file would exceed maximum size allowed.
58	I/O error occurred when resource was zeroed.
59	Replacement file cannot be allocated.
60	Invalid directory entry.
61	Directory and file not on same volume.
62	Reserved.
63	Replacement file is not exclusively allocated to the caller.
64	Out of system space.

Native Mode

Integer Value	Status
65	Cannot allocate FAT/FPT when creating a temporary file.
66	Deallocation error in zeroing file.
67	Resource descriptor destroyed.
68	Invalid resource specification.
69	Error from Resource Management Module (H.REMM).
70	Attempted to modify more than one resource descriptor at same time or attempted to rewrite resource descriptor prior to modifying it.
83	Unable to allocate resource for specified usage.
84	Unable to locate resource (invalid pathname or memory partition definition).
85	Specified access mode not allowed.
86	FPT/FAT space not available.
87	Blocking buffer space not available.
88	Shared Memory Table (SMT) entry not found.
89	Volume Assignment Table (VAT) space not available.
90	Static assignment to dynamic common.
91	Unrecoverable I/O error to volume.
92	Invalid usage specification.
93	Dynamic partition definition exceeds memory limitations.
94	Invalid Resource Requirement Summary (RRS) entry.
95	LFC logically equated to an unassigned LFC.
96	Assigned device not in system.
97	Resource already allocated by requested task.
98	SGO or SYC assignment by real-time task.
99	Common memory conflicts with task's address space.
100	Duplicate LFC assignment attempted.
101	Invalid device specification.
102	Invalid resource id (RID).
103	Specified volume not mounted.
104	J.MOUNT run request failed.
105	Resource is marked for deletion.
106	Assigned device is marked off-line.
107	Segment definition allocation by unprivileged task.

Native Mode

Integer Value	Status
108	Random access not allowed for this access mode.
109	User attempting to open SYC file in a write mode.
110	Resource already opened by this task in a different access mode.
111	Invalid access specification at open.
112	Specified LFC is not assigned to a resource for this task.
113	Invalid allocation index.
114	Close request issued for an unopened resource.
115	Attempt to release an exclusive resource lock that was not owned by this task, or a synchronous lock that was not set.
116	Attempt to release an exclusive resource lock on a resource that has been allocated for exclusive use.
117	Reserved.
118	Attempt to exclude memory partition that is not mapped into requesting task's address space.
119	Reserved.
120	Invalid J.MOUNT request.
121	Timeout occurred while waiting for resource to become available.
122	Reserved.
123	Unable to obtain dual port resource lock (dual port only)
124	Unable to release dual port resource lock (dual port only)
125-132	Reserved.
133	Resource is locked by another task.
134	Shareable resource is allocated by another task in an incompatible access mode.
135	Volume space is not available.
136	Assigned device is not available.
137	Unable to allocate resource for specified usage.
138	Allocated Resource Table (ART) space is not available.
139	Reserved.
140	Volume is not available for a mount with requested usage.
141	Shared Memory Table (SMT) space is not available.
142	Mounted Volume Table (MVT) space is not available.

The following IOSTAT values are returned only with the OPEN, CLOSE, and INQUIRE statements.

Compatible Mode

Integer Value	Status
0	No errors, operation complete.
1	A file of the name specified already exists.
2	A FAST file was specified and collision mapping occurred with an existing directory entry.
3	Restricted access was specified, but no password was entered.
4	Disc space is unavailable.
5	The specified device is not configured.
6	The specified device is offline.
7	A valid restricted file code was not specified.
8	The specified device type is not configured.
9	Invalid username specified.
10	A status of OLD was specified for a nonexistent file.
11	A filename was given for a SCRATCH (temporary) file.
30	Permanent file is exclusively locked.
31	File lock table is full.
32	Nonshared device is busy (already allocated).
33	Disc space is not available.
41	Permanent file is nonexistent.
42	Illegal file password specified.
43	No FAT/FPT space available.
44	No blocking buffer space available.
45	Shared memory table entry not found.
46	Invalid shared memory table password specified.
48	Unrecoverable I/O error to SMD.
49	SGO assignment specified by terminal task.
50	No 'UT' file code exists for terminal task.
51	Invalid RRS entry.
53	Assigned device not on system.
54	Device is in use by requesting task.
55	SGO or SYC assignment by real-time task.
56	Common memory conflicts with allocated task.
57	Duplicate LFC allocation attempted.

Compatible Mode

Integer Value	Status
66	Timeout has occurred while waiting for resource to become available.
74	Attempt to delete a file that does not exist or does not have delete access.

D.2.2 Abort Codes RS and RT

RS-Error Number	Cause of Error
RS01	Error occurred in the routine named in the extended abort code. See the named service for specific reason.
RS02	Error occurred in the routine named in the extended abort code. See the named service for specific reason.
RS03	Error occurred in the routine named in the extended abort code. See the named service for specific reason.
RS04	Error occurred in the routine named in the extended abort code. See the named service for specific reason.
RS05	Error occurred in the routine named in the extended abort code. See the named service for specific reason.
RS06	Error occurred in the routine named in the extended abort code. See the named service for specific reason.
RS07	I/O error while reading load module.
RS08	No free MIDL space.
RS09	Insufficient memory.
RS10	Error occurred in the routine named in the extended abort code. See the named service for specific reason.
RS11	Invalid send buffer address or quantity exceeds 768 bytes.
RS12	Invalid return buffer address.

RS-Error Number	Cause of Error
RS13	Invalid no-wait mode end-action routine address.
RS14	Memory pool unavailable.
RS15	Destination task queue depth exceeded.
RS16	Invalid PSB Address.
RS22	Missing file control block (FCB).
RS29	Request denied, LFC not allocated.
RS30	Request denied, specified LFC not assigned to a permanent disc file.
RS32	Error occurred in the routine named in the extended abort code. See the named service for specific reason.
RS33	Error occurred in the routine named in the extended abort code. See the named service for specific reason.
RS38	Request denied, time out occurred while waiting to become lock owner.
RS47	Invalid time interval request.
RS48	Invalid task number.
RS49	Invalid run request.
RS50	Missing parameter.
RS53	Invalid receiver.
RS60	Invalid address specified.
RS65	Invalid delete request.
RS66	Invalid abort request.
RS67	Invalid resource mark request.
RS68	Taskname/tasknumber not found.
RS69	File control block (FCB) not located.
RS70	Allocation error (appears only if IOSTAT & $N parameters have been omitted).

RS-Error Number	Cause of Error
RS90	Request denied, file lock allocated or exclusively locked.
RS99	An attempt was made to mix calls between SRTL libraries.

RT-Error Number	Cause of Error
RT01	Unformatted read I/O error.
RT02	Formatted read I/O error.
RT03	Unformatted write I/O error.
RT04	Formatted write I/O error.
RT05	Reference made to non-existent device type or address.
RT06	Unit out of range (0-999).
RT07	No left parenthesis in format.
RT08	Transfer index out of range (option 7 or M:ERRFLG can be used to avoid an abort).
RT09	Format error.
RT10	The I/O transfer requirements for the data buffer are incompatible with the amount of available data.
RT11	Format parenthesis level in excess of two.
RT12	Invalid descriptor in format table.
RT13	Argument list exceeds logical read record.
RT14	Incorrect descriptor in format.
RT15	Integer descriptor but non-integer argument (option 7 or M:ERRFLG can be used to avoid an abort).
RT16	Hexadecimal descriptor but non-hexadecimal argument (option 7 or M:ERRFLG can be used to avoid an abort).

RT-Error Number	Cause of Error
RT17	D,E,F,G, descriptor, not real or complex argument (option 7 or M:ERRFLG can be used to avoid an abort).
RT18	Logical descriptor but non-logical argument (option 7 or M:ERRFLG can be used to avoid an abort).
RT19	Attempt to read past EOF/EOM.
RT20	Attempt to write past EOF/EOM.
RT21	Attempt to read past EOF/EOM.
RT22	Attempt to write past EOF/EOM.
RT23	Attempt to backspace following EOF/EOM.
RT24	Rewind after EOF/EOM.
RT25	Formatted record read.
RT26	Unformatted record read.
RT27	Doubleword integer overflow (option 7 or M:ERRFLG can be used to avoid an abort).
RT28	Byte integer input with negative sign (option 7 or M:ERRFLG can be used to avoid an abort).
RT29	Byte integer overflow (option 7 or M:ERRFLG can be used to avoid an abort).
RT30	Halfword integer overflow (option 7 or M:ERRFLG can be used to avoid an abort).
RT31	Fullword integer overflow (option 7 or M:ERRFLG can be used to avoid an abort).
RT32	Illegal character in D,E,F,G input (option 7 or M:ERRFLG can be used to avoid an abort).
RT33	Underflow in floating conversion (option 7 or M:ERRFLG can be used to avoid an abort).
RT34	Overflow in floating conversion (option 7 or M:ERRFLG can be used to avoid an abort).
RT35	Argument list overflow (option 7 or M:ERRFLG can be used to avoid an abort).

RT-Error Number	Cause of Error
RT36	Argument list overflow (option 7 or M:ERRFLG can be used to avoid an abort).
RT37	Not enough arguments were passed.
RT40	Attempt to free busy IOCH/IOCB entry.
RT41	Attempt to link busy IOCH/IOCB entry.
RT42	IOCH/IOCB table overflow.
RT43	ADI wait I/O returned before I/O termination.
RT44	Status parameter not linked to ADI device prior to I/O request.
RT46	ADI table address not on halfword boundary.
RT50	Missing parameter.
RT51	Parameter out of range.
RT52	End of search list reached.
RT53	No unit connection.
RT54	Service is not applicable for SYC.
RT55	Error found in math library routine.
RT60	Illegal random access.
RT61	List-directed I/O (input) encountered, character string split between two records.
RT62	Internal file read/write past EOF/EOM with no end option specified.
RT63	Block number exceeds maximum block number in file.
RT64	Record overflow.
RT65	Record length exceeds maximum allowable.
RT66	Record length not specified for random access or specified for sequential file.
RT67	Implicit open not allowed or random access I/O.
RT68	Reference to sequential operation on a file opened for direct access.

RT-Error Number	Cause of Error
RT69	Error(s) encountered on open.
RT70	File must be unblocked and opened for random access for bufferin/bufferout random I/O.
RT74	Attempt to delete a file that does not exist or does not have delete access.
RT80	Subscript error (i.e., subscript not a decimal number, illegal punctuation, excessive subscripts, or subscript out of range).
RT81	Namelist identifier error (i.e., column 1 non-blank, ampersand character not present, name does not immediately follow ampersand character, or non-blank following name).
RT82	Symbolic name error (no equal sign after variable/array name).
RT83	Data item error (i.e., excessive values for symbol or expected to find symbol).
RT84	Illegal value (i.e., illegal punctuation, missing comma, zero hollerith count, or illegal character in value).
RT85	Attempt to read past EOF/EOM.
RT86	Attempt to write past EOF/EOM.
RT87	Symbolic name not defined in namelist statement.
RT88	Repeat count error.
RT89	Symbolic name exceeds eight characters.
RT90	Invalid read/write operation.
RT91	End-of-file status return pursuant to random access record.
RT92	Random access partition number out-of-range (i.e., partition number not between 1 and 95 inclusive).
RT93	Random access record number out-of-range (i.e., record number not between 1 and 65,535 inclusive).

RT-Error Number	Cause of Error
RT94	Random access transfer length (write/read) or record size definition (define) out-of-range (i.e., transfer record length not between 1 and 65,535 bytes inclusive).
RT95	Invalid random access argument list length.
RT96	FCB table overflow (31) or maximum number of entries allowed in the allocation table (30) has been exceeded.
RT97	Diagnostic output messages exceed 100 lines. To allow more diagnostic messages, statically assign the "DO" file (i.e., $ASSIGN2 DO = SLO, 500).
RT98	Denial return when attempting to allocate file for diagnostic output message.
RT99	Insufficient blocking buffer space (each unit assignment to a system file requires one blocking buffer unless one file is assigned to another, i.e., $AS LFC TO LFC).

The RT prefix can be replaced by a W. or T., which indicates whether the error is warning or terminal, respectively.

D.3 Minor Errors

The following errors are considered minor in the sense that option 7 can be set at runtime or the M:ERRFLG service can be used in the program to avoid an abort.

Error Number
RT08
RT15
RT16
RT17
RT18
RT27
RT28
RT29
RT30
RT31
RT32
RT33
RT34
RT35
RT36

APPENDIX E

COMPARISON OF FORTRAN 77+ AND FORTRAN 77/X32

E.1 General Information

The FORTRAN 77+ and the Scientific Run-time Library (SRTL) products together define both the FORTRAN and the MPX-32 system capabilities of nonbase register FORTRAN. The FORTRAN 77+ compiler is sectioned (i.e., sharable), and not overlaid. It generates object code in CATALOG form that is not sectioned. All data are statically allocated. The compiler FORTRAN 77+ accepts the FORTRAN language as defined by the ANSI X3.9-1978 standard, and supports (in conjunction with SRTL), the MIL-STD-1753, ISA-S61.1 and ISA-S61.2 standards. The language contains several extensions commonly found throughout the industry. The SRTL supplies the I/O and mathematical library support for the generated code.

The FORTRAN 77/X32 and FORTRAN RTL/X32 product consists of a compiler and a library for I/O and mathematical support. The FORTRAN 77/X32 compiler is overlaid and not sectioned. It produces an object module acceptable to Linker/X32 that is sectioned. Data are allocated both statically and on a stack. The FORTRAN 77/X32 compiler accepts the FORTRAN language as defined by the ANSI X3.9-1978 standard, and with the support library, satisfies most of the MIL-STD-1753 standard and the bit manipulation functions of the ISA-S61.1 standard. FORTRAN 77/X32 contains some of the extensions found in FORTRAN 77+.

E.2 Similarities Between FORTRAN 77+ and FORTRAN 77/X32

As both FORTRAN products are based on the ANSI X3.9-1978 standard for the FORTRAN programming language, the greatest similarities between the two language products are found in those features specified in the ANSI X3.9-1978 standard. The following language features are syntactically and semantically identical in both products:

- Control Statements
  - Block IF-THEN-ELSE-ELSE IF-END IF
  - Logical and arithmetic IF
  - DO
  - Assigned, computed, simple GO TO
  - CALL and RETURN
  - CONTINUE, STOP, PAUSE, and END

- Data Specification Statements
  - DIMENSION
  - COMMON and EQUIVALENCE
  - IMPLICIT and PARAMETER
  - EXTERNAL and INTRINSIC
  - SAVE and DATA
  - INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER
  - INTEGER*1 and INTEGER*2
  - NAMELIST

- Subprogram Statements
  SUBROUTINE, FUNCTION, PROGRAM, ENTRY, and BLOCK DATA

- Other Statements
  Assignment
  Statement function
  FORMAT

- Other Features
  Intrinsic functions
  Expressions involving standard data types
  Arrays, substrings, and dummy arguments

In the area of input/output, ANSI X3.9-1978 is a permissive standard; that is, it allows wide latitude in defining the underlying system while specifying a language-level I/O model which can be used to write portable software. Both FORTRAN products satisfy the language-level I/O model and both support the syntax of the statements listed below.

READ
WRITE
PRINT
OPEN
CLOSE
INQUIRE
ENDFILE
BACKSPACE
REWIND
NAMELIST

However, I/O implementation restrictions and I/O system-level capabilities differ.

In addition to the features of ANSI X3.9-1978, both products support the following features, although some differences in syntax, semantics, and limitations may exist:

- INCLUDE of source text
- Symbolic name extensions
  Longer than six characters
  Underline allowed in name
- Hollerith data type
- Text form extensions (number of continuation lines, trailing comments, conditional compilation, multiple statements per line)
- IMPLICIT NONE statement
- Compiler directive statements (OPTION, PAGE, SPACE)
- DO statement extensions (DO WHILE, END DO, LEAVE, DO FOREVER)
- Debugging support (generated code listing and local symbol support)
- Operating system service entry points (basic I/O)
- Arrayed statement extensions (array as target of assignment statement)
- Intrinsic function extensions (ANSI 8X proposals and MIL-STD-1753 features)
- Hexadecimal and octal constants in MIL-STD-1753 forms

- Data structuring features
  - Boundary alignment of data in COMMON
  - Mixing character and noncharacter data in COMMON
  - EQUIVALENCE - character and noncharacter
- Datapool support

E.3 Differences Between FORTRAN 77+ and FORTRAN 77/X32

The major differences between the FORTRAN 77+ and FORTRAN 77/X32 products are in the support for data types, I/O, control flow constructs, and system-level extensions (e.g., extended memory).

The differences are more syntactic than functional. For example, specific extended memory support is provided in FORTRAN 77+, whereas the large memory address space of base mode is inherent in the design of the FORTRAN 77/X32 product. Where two features provide redundant capability (e.g., Hollerith and CHARACTER, ENCODE/DECODE and internal I/O) only the ANSI X3.9-1978 compatible feature is provided in FORTRAN 77/X32.

Note that the FORTRAN 77+ and FORTRAN 77/X32 compiler options are not equivalent (e.g., the meaning of option 19 differs between the two compilers).

The following features are present in FORTRAN 77+ but are not supported in FORTRAN 77/X32.

- Extended Data Type Support (Implied is support for the IMPLICIT statement, I/O statements, constants, operators, and conversions for each extended data type.)
  - INTEGER*8
  - LOGICAL BIT, LOGICAL*1
  - COMPLEX*16 (Double Precision Complex)

- I/O Extensions
  - SKIPFILE, BACKFILE, ACCEPT, TYPE, PUNCH statements
  - Auxiliary I/O keywords

- Control Flow Constructs
  - DO UNTIL
  - SELECT CASE
  - Internal procedures (subroutines and functions)

- System Features - functionally equivalent but syntactically different

- Miscellaneous Intrinsic Function Extensions
  - LOCF

- Miscellaneous Features
  - Integer character range for ICHAR: 0-127
  - After PAUSE command, CONTINUE reactivates task in interactive mode; !CONTINUE reactivates task in batch mode

- FORTRAN-66+ Oriented Features (Obsolete)
  - Extended DO Loop range
  - Single trip DO Loop option
  - ENCODE/DECODE
  - Reduced dimensionality in array subscripts

- Symbolic Name Extensions
  - 8 significant characters recognized
  - Colon allowed in name

- Data Statement Extensions
  - Use of Hollerith (beyond ANSI X3.9-1978)
  - Use of CHARACTER (beyond ANSI X3.9-1978)

- Hollerith and CHARACTER Extensions
  - Hollerith beyond ANSI X3.9-1978 Appendix
  - CHARACTER quote convention for non-printing characters

- Other Forms of Constants
  - Hexadecimal constants: $X'd_1...d_n'$ or $nZd_1...d_n$
  - Binary constants: $B'd_1...d_n'$

- Miscellaneous Extensions
  - Inline assembly code
  - Missing actual arguments in subroutine/function calls
  - Multiple targets in assignment statement
  - Use of alternate returns as labeled entry points (as used in SRTL)
  - INTEGER constant data type determined by magnitude

The following features are supported in the FORTRAN 77/X32 product but are not in the FORTRAN 77+ product.

- System Features
  - Sharable code
  - Use of stack default for data (i.e., default is not SAVE)
  - Subroutine and function interface checking in conjunction with the linker, LINKER/X32
  - Use of GLOBAL COMMON provided by the linker (LINKER/X32), not by the provision of a GLOBAL COMMON statement

- Miscellaneous Intrinsic Function Extensions (beyond ANSI X3.9-1978)

EVEN	FRACTION	VERIFY	ADJUSTR	CEILING
ODD	EXPONENT	ADJUSTL	FLOOR	

- Symbolic Name Extensions
  - 16 significant characters recognized

- Miscellaneous Features
  - Integer character range for ICHAR: 0-255
  - After PAUSE command, carriage return from operator's console reactivates task

E.4 I/O Differences

Table E-1 summarizes the differences between FORTRAN 77/X32 and FORTRAN 77+ in support for I/O features.

Table E-1
I/O Features (Sheet 1 of 2)

Feature	How supported	
	FORTRAN 77/X32	FORTRAN 77+
Extended keywords		
UNIT	Value must be a non-negative integer expression in the range 1-999.	Value must be a 3-character integer (0-999), or a 1- to 3-character string, left-justified and blank-filled in an INTEGER variable.
FILE	May be an MPX-32 pathname or an MPX-32 pathname followed by fields used in an MPX-32 ASSIGN line.	Must be an MPX-32 pathname.
BLOCKED	Pathname field (BLOCKED=)	Individual keywords in OPEN and INQUIRE.
OPENMODE	Pathname field (ACCESS=)	
READONLY	Pathname field (ACCESS=)	
SPOOLFILE	Autoconnect	
VOLUME	Pathname field	
ALTUNIT	Pathname field (LFC=)	
DEVICE	Pathname field (DEV=)	
File creation attributes	CREATE string in OPEN.	Individual keywords in OPEN.
List-directed I/O		
Input	No differences.	No differences.
Output	Format differences; multiple succeeding values are emitted with multiplicity; i.e., N* value	
Preconnections	Static assignments of units 1-999. Autoconnect units as specified in Table E-2	Uses MPX-32 LFCs.

Table E-1
I/O Features (Sheet 2 of 2)

Feature	How supported	
	FORTRAN 77/X32	FORTRAN 77+
Formats	No Hollerith editing; No hexadecimal editing; D editing emits exponents as 'E±n'	Hollerith editing; Hexadecimal editing; D editing emits exponents as 'D±n'
Scale factor	-128 to +127	-77 to +76
Forms Control	0, 1, +, - >turn headers on <turn headers off = clear header this page	0, 1, +, - (MPX-32, Release 1.x). (For MPX-32, Release 2.x and 3.x, same as FORTRAN 77/X32.)

Table E-2
FORTRAN/X32 Autoconnect Units

Unit	Equivalent MPX-32 Assignment	File	Usage
1	ASSIGN 1 to LFC=UT	User terminal	Input or output
2	ASSIGN 2 to LFC=UT	User terminal	Input or output
5	ASSIGN 5 to SYC	System SYC	Input only
6	ASSIGN 6 to SLO	System SLO	Output only
11	ASSIGN 11 to SGO	System SGO	Input or output
12	ASSIGN 12 to SBO	System SBO	Input or output

INDEX

A editing, 12-16
  noncharacter data types, 12-18
Abort Codes - FT, D-3/4
Abort Codes - RS and RT, D-11/17
ABS intrinsic function, 9-7
ACCEPT statement, 11-8, 11-12
ACCESS specifier, 11-28, 11-44
ACOS intrinsic function, 9-17
ADDR intrinsic function, 9-11
AIMAG intrinsic function, 9-7
AINT intrinsic function, 9-7
ALLOCATE specifier, 11-28, 11-44
ALOG intrinsic function, 9-11
ALOG10 intrinsic function, 9-11
Alphabetic characters, 2-6
Alphanumerics, 2-7
Alternate returns, 9-28/30
ALTUNIT specifier, 11-29, 11-38
AMAX0 intrinsic function, 9-12
AMAX1 intrinsic function, 9-12
AMIN0 intrinsic function, 9-12
AMIN1 intrinsic function, 9-12
AMOD intrinsic function, 9-12
AND intrinsic function, 9-15
ANINT intrinsic function, 9-7
Apostrophe editing, 12-17
Arithmetic assignment
  statements, 5-2/4
Arithmetic expressions
  definition, 4-1
  evaluation, 4-6/7
  in assignment statements, 5-2/4
  rules for constructing, 4-1/3
  type determination, 4-4/5
Arithmetic IF statement, 6-5
Arithmetic operators, 4-1
Arrays
  and EQUIVALENCE, 7-15/23
  assignment statements, 5-8
  declarators, 3-15, 7-2
  definition, 3-14
  dimensions, 3-15/16, 7-2
  elements of, 3-16
  establishing, 3-14
  in specification statements, 7-1/26
  processing in subprograms, 9-24
  storage, 3-17/18, 7-15/16
ASCII code set, C-1/7
ASIN intrinsic function, 9-17
ASSIGN statement, 5-6
Assigned GO TO statement, 6-4
Assignment statements
  arithmetic, 5-2/4
  ASSIGN statement, 5-6
  character, 5-5
  full array, 5-8
  general considerations, 5-1
  logical, 5-4
  multiple, 5-7
Asterisk as format identifier, 11-5,
  11-24
Asynchronous input/output, 11-59/61
ATAN intrinsic function, 9-17
ATAN2 intrinsic function, 9-17
Auxiliary input/output statements
  BACKFILE, 11-52/53
  BACKSPACE 11-51/52
  CLOSE, 11-39/40
  ENDFILE, 11-54/55
  INQUIRE, 11-41/43
  OPEN, 11-27
  REWIND, 11-56/57
  SKIPFILE, 11-53/54

BACKFILE specifiers
  ERR, 11-52
  IOSTAT, 11-53
BACKFILE statement, 11-52/53
BACKSPACE specifiers
  ERR, 11-51
  IOSTAT, 11-51
BACKSPACE statement, 11-51/52
Binary constants, 3-12
Bit data type, 3-8
Blank character
  in symbolic names, 3-13
BLANK specifier, 11-29, 11-44, 12-4,
BLOCK DATA statement, 10-1/2
BLOCK DATA subprograms, 10-1/2
BLOCKED specifier, 11-29, 11-44
Block IF statement
  construct, 6-7
  ELSE, 6-11/12
  ELSE IF THEN, 6-9/11

END IF, 6-12
IF level, 6-14
nested, 6-13
BN descriptor, 12-3/4, 12-15
BTEST intrinsic function, 9-15
BUFFERIN subroutine, 11-59/61
BUFFEROUT subroutine, 11-59/61
BZ descriptor, 12-3/4, 12-15

CABS intrinsic function, 9-7
CALL statement, 9-22/23
CALL STATUS service, 11-61
Calling a FORTRAN subroutine from assembly language programs, 15-17/18
Calling assembler routines, 15-7/18
CASE statement, 6-26/27
Cataloging a FORTRAN program, 15-4
CCOS intrinsic function, 9-17
CDABS intrinsic function, 9-7
CDCOS intrinsic function, 9-17
CDEXP intrinsic function, 9-9
CDLOG intrinsic function, 9-11
CDSIN intrinsic function, 9-18
CDSQRT intrinsic function, 9-13
CEXP intrinsic function, 9-9
CHAR intrinsic function, 9-8
Character
assignment statements, 5-5
constants, 3-9, 8-2
data type, 3-9
expressions, 4-7/10
format specifications, 12-16/17
operator, 4-7
relational expressions, 4-10/11
set, 2-6
strings, 3-9
substrings, 4-8/9
Character editing
A editing, 12-16
Apostrophe editing, 12-17
H editing, 12-17
R editing, 12-19
Character expression
definition, 4-7/8
in assignment statements, 5-5
operand, 4-7
primaries, 4-7
relational, 4-10/11
substrings, 4-8/9
CHARACTER statement, 7-6/7
CLEAR specifier, 11-30
CLOG intrinsic function, 9-11
CLOSE statement, 11-39
CLOSE statement specifiers
ERR, 11-39
IOSTAT, 11-39
STATUS, 11-39
CMPLX intrinsic function, 9-8
Collating sequence, 2-6
Colon descriptor, 12-27
Comment lines, 2-5
Common block storage, 7-12
Common JCL statements used with FORTRAN, 15-3
COMMON statement, 7-11/13
Compile time diagnostics, D-1
Compiler options, 15-1/3
Compiler parameter lists, 15-19
Compiling, cataloging, and executing a FORTRAN program, 15-4/5
Complex constants
complex doubleword, 3-8
complex word, 3-7
Complex editing
I editing, 12-12
Z editing, 12-13/14
Complex constants, 3-7/8
Complex data types, 3-6/7
COMPLEX statement, 7-4, 7-8
Complex word data type, 3-7
Computed GO TO statement, 6-3
Concatenation, 4-7
CONJG intrinsic function, 9-8
Constants
binary, 3-12
character, 3-9, 8-3
character versus Hollerith, 3-10
complex, 3-7/8
double precision, 3-5/6
hexadecimal, 3-11/12
Hollerith, 3-10/11
integer, 3-2
logical or bit, 3-8
octal, 3-12
real, 3-3/4
Context errors, D-1
CONTIGUOUS specifier, 11-30, 11-45
CONTROLBITS specifier, 11-30
Continuation field, 2-2
Continuation lines and logical IF, 6-6
CONTINUE statement, 6-24
Control and interpretation of data, 11-7/8
Control features of FORMAT statements, 12-22/28
Control information list
definition, 11-4
END=, 11-4

ERR=, 11-4
FMT=, 11-4
IOSTAT=, 11-4
REC=, 11-4
UNIT=, 11-4
Control statements
arithmetic IF, 6-5
assigned GO TO, 6-4
block IF, 6-7
CALL, 9-22/23
computed GO TO, 6-3
CONTINUE, 6-24
definition, 6-1
DO, 6-15
DO forever, 6-20
DO UNTIL, 6-21
DO WHILE, 6-22
ELSE, 6-11/12
ELSE IF THEN, 6-9/11
END, 6-28
END DO, 6-24
END IF, 6-12
END SELECT, 6-27
IF THEN, 6-8/9
LEAVE, 6-23
logical IF, 6-6
PAUSE, 6-29
RETURN, 9-22
SELECT CASE, 6-25/26
STOP, 6-28
unconditional GO TO, 6-2
COS intrinsic function, 9-17
COSH intrinsic function, 9-18
CSIN intrinsic function, 9-18
CSQRT intrinsic function, 9-13

D editing
and repeat specification, 12-24/25
and scale factor, 12-22/24
description, 12-7
DABS intrinsic function, 9-7
DACOS intrinsic function, 9-17
DASIN intrinsic function, 9-17
Data Conversion, 8-1
DATA statement, 8-1/6
Data transfer input/output statements
ACCEPT, 11-8, 11-12
PRINT, 11-13,11-16/17
PUNCH, 11-13, 11-17
READ, 11-8, 11-11/12
TYPE, 11-13, 11-17/18
WRITE, 11-13, 11-16/17
Data types
character, 3-9
complex, 3-6/7
double precision, 3-5
for list-directed input, 11-25/26
for list-directed output, 11-26/27
for list-directed transfer, 11-26
integer, 3-1/2
logical, 3-8
order of precedence, 4-3
real, 3-3
DATAN intrinsic function, 9-17
DATAN2 intrinsic function, 9-17
DATAPOOL statement, 7-14/15
DBLE intrinsic function, 9-8
DCMPLX intrinsic function, 9-9
DCONJG intrinsic function, 9-8
DCOS intrinsic function, 9-17
DCOSH intrinsic function, 9-18
DDIM intrinsic function, 9-9
DECODE statement, 11-57/59
DENSITY specifier, 11-30
DEVICE specifier, 11-30, 11-45
Device type codes, A-4
DEXP intrinsic function, 9-9
Diagnostics
abort codes - FT, D-3/4
abort codes - RS and RT, D-11/17
compile time, D-1
context errors, D-1
execution time, D-4A
iostat values, D-5/11
minor errors, D-18
source line errors, D-1
DIM intrinsic function, 9-9
DIMAG intrinsic function, 9-7
Dimension range, -3-15/16
DIMENSION statement, 7-2/3
DINT intrinsic function 9-7
Direct access input/output, 11-3
DIRECT specifier, 11-45
Disc files for output data, 15-6
Disc sector size, A-1
DLOG intrinsic function, 9-11
DLOG10 intrinsic function, 9-11
DMAX1 intrinsic function, 9-12
DMIN1 intrinsic function, 9-12
DMOD intrinsic function, 9-12
DNINT intrinsic function, 9-7
DO
forever, 6-20
index of, 6-17
iteration control, 6-18/19
loops and option 11, 6-18
nested loops, 6-18/19
statement, 6-15

terminal statement of, 6-17
transfer of control, 6-19
UNTIL statement, 6-21
WHILE statement, 6-22
Documentation conventions, 1-5/6
Double precision constants, 3-5/6
Double precision data type, 3-5
Double precision editing, 12-7
DOUBLE PRECISION statement, 7-3, 7-8
DPROD intrinsic function, 9-9
DREAL intrinsic function, 9-13
DSIGN intrinsic function, 9-13
DSIN intrinsic function, 9-18
DSINH intrinsic function, 9-18
DSQRT intrinsic function, 9-13
DTAN intrinsic function, 9-18
DTANH intrinsic function, 9-18
Dummy arguments
in functions and subroutines, 9-1/2
in internal procedures, 9-24

E editing
and repeat specification, 12-24/25
and scale factor, 12-22/24
description, 12-9
Edit descriptors, 12-3/4
Editing
character editing, 12-16/19
complex, 12-12
logical, 12-20
numeric, 12-7/15
ELSE IF THEN statement, 6-9/11
ELSE statement, 6-11/12
ENCODE statement, 11-57/59
END DO statement, 6-24
ENDI statement, 14-1
END IF statement, 6-12
END INTERNAL statement, 9-25
END SELECT statement, 6-27
END specifier, 11-4, 11-9, 11-14
END statement, 6-28
ENDFILE specifiers
ERR, 11-55
IOSTAT, 11-55
ENDFILE statement, 11-54/55
End-of-file detection, A-2
ENTRY association, 9-27
ENTRY statement, 9-20/26
EQUIVALENCE
and arrays, 7-17/19
and boundaries, 7-16/17
and common interaction, 7-22/23
and substrings, 7-20/21
statement, 7-15/16

ERR specifier, 11-4, 11-31, 11-45
Evaluation of expressions
arithmetic, 4-6/7
logical, 4-12
Exclamation mark, 2-7
Executable statements
arithmetic IF, 6-5
ASSIGN, 5-6
assigned GO TO, 6-4
assignment, 5-1/8
auxiliary input/output, 11-27/56
BACKFILE, 11-52/53
BACKSPACE, 11-51
block IF, 6-7
CALL, 9-22/23
CASE, 6-26
character assignment, 5-5
CLOSE, 11-39/40
computed GO TO, 6-3
CONTINUE, 6-24
control, 6-1/29
definition of, 2-3
DO, 6-15
DO forever, 6-20
DO UNTIL, 6-21
DO WHILE, 6-22
ELSE, 6-11/12
ELSE IF THEN, 6-9/11
END, 6-28
END DO, 6-24
END IF, 6-12
END SELECT, 6-27
ENDFILE, 11-54/55
INQUIRE, 11-41/43
LEAVE, 6-23
logical IF, 6-6
OPEN, 11-27/28
PAUSE, 6-29
PRINT, 11-13
READ, 11-8
RETURN, 9-27/28
REWIND, 11-56/57
SELECT CASE, 6-25/26
SKIPFILE, 11-53/54
STOP, 6-28
unconditional GO TO, 6-2
WRITE, 11-13
Executing a FORTRAN program, 15-4
Execution-time diagnostics, D-4A
EXIST specifier, 11-45
EXP intrinsic function, 9-9
Explicit CHARACTER statement, 7-6/7
Explicit type statements, 7-4/6
Exponentiation, 4-1, 4-6
Expressions

arithmetic, 4-1/6
character, 4-7/9
character constant, 4-8
character substring, 4-8/9
in assignment statements, 5-5
logical, 4-12/14
relational, 4-10/11
use of hexadecimal, binary, and octal constants in, 4-17
use of Hollerith constants in, 4-15/16
type determination, 4-4/5
EXTEND specifier, 11-31, 11-45
Extended addressing, 13-1/5
EXTENDED BASE statement, 13-2
EXTENDED BLOCK statement, 13-1
EXTENDED DUMMY statement, 13-2
Extended memory restrictions, 13-4
EXTENDIBLE specifier, 11-31, 11-46
External files, 11-2/3
External functions, 9-1
EXTERNAL statement, 7-23/25

F editing
and repeat specification, 12-24/25
and scale factor, 12-22/24
description, 12-10
Field separators, 12-26/27
File assignments, 15-1
File positioning input/output statements
BACKFILE, 11-52/53
BACKSPACE, 11-51/52
REWIND, 11-56
SKIPFILE, 11-53/54
FILE specifier, 11-32
File system
compatible mode, 1-1
native mode, 1-1
Files
access methods, 11-3
blocked, 11-3/4
external, 11-2
internal, 11-2
record position within, 11-3
unblocked, 11-3/4
FILESIZE specifier, 11-32, 11-46
FLOAT intrinsic function, 9-13
FMT specifier, 11-4
FORM specifier, 11-32, 11-46
Format
control list specification and record demarcation, 12-5/6
specifications expressed as character constants, 12-2
specification methods, 12-1
specifications stored in variables and arrays, 12-2
FORMAT statement, 12-1
FORMAT statement control features
colon descriptor, 12-28
field separators, 12-26
repeat specification, 12-24/25
scale factor, 12-22/24
Formatted
input statements, 11-8/12
records, 11-1
output statements, 11-13/18
FORMATTED specifier, 11-46
Forms control on output, 12-6/7
FORTRAN
and job control language, 15-3
character set, 2-7
compiler options, 15-1/3
compile-time diagnostics, D-1/2
execution-time diagnostics, D-5
program, 2-1/12
run-time options, 15-3
statements, 2-3/5
FORTRAN 77+ Release 4.1 Versus Release 4.0, 1-3
FORTRAN statements
ACCEPT, 11-8, 11-12
ASSIGN, 5-6
BACKFILE, 11-52/53
BACKSPACE, 11-51/52
BIT, 7-5
BLOCK DATA, 10-1/2
CALL, 9-22/23
CHARACTER, 7-6/7
CLOSE, 11-39/40
COMMON, 7-11/13
COMPLEX, 7-4, 7-8
CONTINUE, 6-24
DATA, 8-1/6
DECODE, 11-57/59
DIMENSION, 7-2/3
DO, 6-15
DO forever, 6-20
DO UNTIL, 6-21
DO WHILE, 6-22
DOUBLE PRECISION, 7-3, 7-8
ELSE, 6-11/12
ELSE IF THEN, 6-9/11
ENCODE, 11-57/59
END, 6-28
END DO, 6-24
END IF, 6-12
END SELECT, 6-27
ENDFILE, 11-49/55

ENTRY, 9-20/26
EQUIVALENCE, 7-15/16
EXTENDED BASE, 13-2
EXTENDED BLOCK, 13-1
EXTENDED DUMMY, 13-2
EXTERNAL, 7-23/25
FORMAT, 12-1
FUNCTION, 9-19
GO TO, 6-2/4
IF, 6-5/14
IMPLICIT, 7-8/9
IMPLICIT NONE, 7-9
INCLUDE, 2-7/9
INQUIRE, 11-41/44
INTEGER, 7-4, 7-8
INTRINSIC, 7-25/26
LEAVE, 6-23
LOGICAL, 7-4, 7-8
NAMELIST, 11-19
OPEN, 11-27/28
OPTION, 2-10/11
PAGE, 2-9
PARAMETER, 7-10
PAUSE, 6-29
PRINT, 11-13
PROGRAM, 2-1
PUNCH, 11-13
READ, 11-8
REAL, 7-5, 7-8
RETURN, 9-27/28
REWIND, 11-56/57
SAVE, 7-26/27
SELECT CASE, 6-25/26
SKIPFILE, 11-53/54
SPACE, 2-11
STOP, 6-28
SUBROUTINE, 9-21/22
TYPE, 11-13, 11-17/18
USER, 2-11/12
WRITE, 11-13, 11-16/18
Full array assignments, 5-8
Function reference appearing in I/O statements, 11-1
FUNCTION statement, 9-19
Function subprograms, 9-19/20

G editing
and repeat specification, 12-24/25
and scale factor, 12-22/24
description, 12-11
Generated code listings, B-9/10
GLOBAL COMMON statement, 7-13
GO TO statements
assigned, 6-4
computed, 6-3
unconditional, 6-2
GOULD CSD FORTRAN 77+ Versus ANSI X3.9-1978, 1-3/5

H editing, 12-17
Hexadecimal constants
definition, 3-11
in expressions, 4-17
in DATA statement, 8-2
Hexadecimal type code, 15-15
HFIX intrinsic function, 9-6
Hollerith constants
in expressions, 4-15/16
versus character constants, 3-10
Hollerith strings in argument lists, 4-18

IABS intrinsic function, 9-7
IAND intrinsic function, 9-15
IBCLR intrinsic function, 9-15
IBITS intrinsic function, 9-15
IBSET instrinsic function, 9-15
ICHAR intrinsic function, 9-10
Identification field, 2-3
IDIM intrinsic function, 9-9
IDINT intrinsic function, 9-10
IDNINT intrinsic function, 9-12
I editing
and repeat specification, 12-24/25
description, 12-12
IEOR intrinsic function, 9-15
IF statements
arithmetic, 6-5
block, 6-7
logical, 6-6
IF block, 6-7/14
IF level, 6-14
IFIX intrinsic function, 9-10
IMPLICIT NONE statement, 7-9
IMPLICIT type statement, 7-8/9
Implicit typing convention, 3-13/14
Implied DO in DATA statement, 8-4/6
INCLUDE directive, 2-7/9
INCREMENT specifier, 11-32, 11-46
INDEX intrinsic function, 9-10
Index of the DO, 6-17
Initial line, 2-6
Inline assembly language coding, 14-1/7
argument field, 14-1
argument field directives, 14-4

comment field, 14-1
general instruction format, 14-2
immediate operand instructions, 14-3
interregister instructions, 14-3
label field, 14-1
memory bit and condition code instructions, 14-3
memory reference instructions, 14-2
operation control instructions, 14-4
operation field, 14-1
referencing dummy variables, 14-6
referencing variables in extended memory, 14-7
referencing variables in local storage, 14-6
INLINE directives
AC, 14-5
BOUND, 14-5
DATA, 14-4
EQU,14-6
GEN, 14-5
RES, 14-6
INLINE statement, 14-1
Input/output definition, 11-5, A-1
multiple data identifiers, 11-6/7
single datum identifier, 11-6
Input/output files, 11-2
Input/output statements
auxiliary, 11-27/56
BACKFILE, 11-52/53
BACKSPACE, 11-51/52
data transfer, 11-8/15
ENDFILE, 11-54/55
file positioning, 11-51/56
function reference appearing in, 11-1
READ, 11-8/12
REWIND, 11-56/57
SKIPFILE, 11-53/54
WRITE, 11-13/18
Input/output records, 11-1/2
Input statements, 11-8/12
Input using NAMELIST, 11-19/24
INQUIRE by file statements, 11-41/42
INQUIRE by file, native mode, 11-41
INQUIRE by file, compatible mode, 11-41/42
INQUIRE by unit statements, 11-42/43
INQUIRE by unit, native mode, 11-42
INQUIRE by unit, compatible mode, 11-43
INQUIRE statement specifiers
ACCESS, 11-44
ALLOCATE, 11-44
BLANK, 11-44
BLOCKED, 11-44
CONTIGUOUS, 11-45
DEVICE, 11-45
DIRECT, 11-45
ERR, 11-45
EXIST, 11-45
EXTEND, 11-45
EXTENDIBLE, 11-46
FILESIZE, 11-46
FORM, 11-46
FORMATTED, 11-46
INCREMENT, 11-46
IOSTAT, 11-46
MAXSIZE, 11-46
MININCREMENT, 11-46
NAME, 11-46
NAMED, 11-46
NEXTREC, 11-47
NUMBER, 11-47
OPENED, 11-47
OPENMODE, 11-47
OTHERACESS, 11-47
OWNERACCESS, 11-47
PROJECTACCESS, 11-47
QUEUE, 11-48
QUEUED, 11-48
READONLY, 11-48
RECL, 11-48
SEQUENTIAL, 11-48
SHARED, 11-48
SPOOLFILE, 11-48
UNFORMATTED, 11-48
INT intrinsic function, 9-10
Integer constants, 3-2
Integer data types, 3-1/2
Integer editing, 12-12
INTEGER statement, 7-4, 7-8
Internal files, 11-2
Internal procedures
and dummy arguments, 9-25
INTERNAL FUNCTION, 9-24
INTERNAL SUBROUTINE, 9-24
referencing, 9-25
Interpretation of blanks on input, 12-4
Intrinsic functions, 9-5/6
INTRINSIC statement, 7-25/26
IOR intrinsic function, 9-16
IOSTAT specifier, 11-4, 11-8/10 11-13/14, 11-32, 11-46
IOSTAT values, D-5/11
ISHFTC intrinsic function, 9-16
ISHIFT intrinsic function, 9-16
ISIGN intrinsic function, 9-13

Job control for batch jobs, 15-1

KEY specifier, 11-32

L editing
  and repeat specification, 12-24/25
  description, 12-20
LEAVE statement, 6-23
LEN intrinsic function, 9-11
LGE intrinsic function, 9-14
LGT intrinsic function, 9-14
Lines, 2-5
List-directed formatting,
  definition, 11-24
  input, 11-25/26
  output, 11-26/27
LLE intrinsic function, 9-13
LLT intrinsic function, 9-13
LOCF intrinsic function, 9-11
LOG10 intrinsic function, 9-11
Logical assignment statement, 5-4
Logical data types, 3-8
Logical editing, 12-20
Logical expressions
  definition, 4-12
  evaluation of, 4-14
  operators, 4-12/14
  rules for constructing, 4-14
Logical file code assignments, 15-1
Logical IF statement, 6-6
Logical operators, 4-12/14
Logical or bit constants, 3-8
Logical record length, A-3
LOGICAL statement, 7-4, 7-8

Main program, 2-1
MAX0 intrinsic function, 9-12
MAX1 intrinsic function, 9-12
Maximum record length, A-2
MAXSIZE specifier, 11-33, 11-46
MININCREMENT specifier, 11-33, 11-46
MIN0 intrinsic function, 9-12
MIN1 intrinsic function, 9-12
Minor errors, D-18
Mismatched argument lists, 9-32
MOD intrinsic function, 9-12
Modes of installation, 1-1/2
MPX-32
  datapool, 7-14/15
  extended addressing, 13-1/5
  EXTENDED BASE statement, 13-2
  EXTENDED BLOCK statement, 13-1
  EXTENDED DUMMY statement, 13-2
  extended memory restrictions, 13-4
  global common, 7-14
  input/output terms, A-1
  maximum sixes, A-2/3
  observations, A-2
Multiple assignment statements, 5-7
Multiple statement, 2-5

NAME specifier, 11-46
NAMED specifier, 11-46
NAMELIST
  data items in input record, 11-21
  input from user terminal, 11-20
  input from other than user terminal, 11-20
  output data formats, 11-22/24
  statement, 11-19
Nested block IF construct, 6-13
Nested DO loops, 6-18/19
NEXTREC specifier, 11-47
NINT intrinsic function, 9-12
Nonexecutable statements
  BLOCK DATA, 10-1/2
  COMMON, 7-11/13
  DATA, 8-1/6
  DIMENSION, 7-2/3
  END INTERNAL, 9-25
  ENDI, 14-1
  ENTRY, 9-20
  EQUIVALENCE, 7-15/16
  EXTENDED BASE, 13-2
  EXTENDED BLOCK, 13-1
  EXTENDED DUMMY, 13-2
  EXTERNAL, 7-23/25
  FORMAT, 12-1
  FUNCTION, 9-19
  IMPLICIT, 7-8/9
  IMPLICIT NONE, 7-9
  INCLUDE, 2-7/9
  INLINE, 14-1
  INTERNAL FUNCTION, 9-24
  INTERNAL SUBROUTINE, 9-24
  INTRINSIC, 7-25/26
  NAMELIST, 11-19
  OPTION, 2-9/10
  PAGE, 2-10/11
  PARAMETER, 7-10
  PROGRAM, 2-1
  SAVE, 7-26/27
  SPACE, 2-11
specification statements, 7-1/27
statement function statement, 9-3/4
  SUBROUTINE, 9-21/22

USER, 2-11/12
NOT intrinsic function, 9-16
No-wait input/output, A-7
NUMBER specifier, 11-47
Numeric characters, 2-7
Numeric editing
and blank interpretation, 12-4
and plus sign control, 12-14
and repeat specification, 12-24/25
and scale factors, 12-22/24
complex editing, 12-12
D editing, 12-7
E editing, 12-9
F editing, 12-10
G editing, 12-11
I editing, 12-12
Z editing, 12-13

Octal constants, 3-12
OPEN statement, 11-27/28
OPEN statement specifiers
ACCESS, 11-28
ALLOCATE, 11-28
ALTUNIT, 11-29
BLANK, 11-29
BLOCKED, 11-29
CLEAR, 11-30
CONTIGUOUS, 11-30
CONTROLBITS, 11-30
DENSITY, 11-30
DEVICE, 11-30
ERR, 11-31
EXTEND, 11-31
EXTENDIBLE, 11-31
FILE, 11-32
FILESIZE, 11-32
FORM, 11-32
INCREMENT, 11-32
IOSTAT, 11-32
KEY, 11-32
MAXSIZE, 11-33
MININCREMENT, 11-33
OPENMODE, 11-33
OTHERACCESS, 11-33
OWNERACCESS, 11-33
PASSWORD, 11-34
PROJECT, 11-34
PROJECTACCESS, 11-34
QUEUE, 11-34
READONLY, 11-34
RECL, 11-35
REEL, 11-35
SHARED, 11-35
SPOOLFILE, 11-35
START, 11-35
STATUS, 11-35/36
UNIT, 11-36
USER, 11-36
VOLUME, 11-37
WAIT, 11-37
OPENED specifier, 11-47
OPENMODE specifier, 11-33, 11-47
OPTION directive, 2-9/10
Order of statements, 2-4
OTHERACCESS specifier, 11-33, 11-47
Output statements, 11-13/18
Output using NAMELIST, 11-19/24
OWNERACCESS specifier, 11-33, 11-47

PAGE directive, 2-10/11
Parameter lists generated
by the compiler, 15-18
PARAMETER statement, 7-10
PASSWORD specifier, 11-34
PAUSE statement, 6-29
Physical byte count, A-3
PRINT statement, 11-13, 11-16/17
PROGRAM statement, 2-1
Program units 2-1
PROJECT specifier, 11-34
PROJECTACCESS specifier, 11-34,
11-47
PUNCH statement, 11-13, 11-17

QUEUE specifier, 11-34, 11-48
QUEUED specifier, 11-48

R editing
and repeat specification, 12-24/25
description, 12-19
READ statement, 11-8, 11-11
READONLY specifier, 11-34, 11-48
Real constants, 3-4
Real data type, 3-3
REAL intrinsic function, 9-13
REAL statement, 7-5, 7-8
REC specifier, 11-4
RECL specifier, 11-35, 11-48
Record
demarcation, 12-5
endfile, 11-1/2
formatted, 11-1/2

unformatted, 11-1/2
REEL specifier, 11-35
Relational expressions
character, 4-11
definition, 4-10
operators within, 4-10
Relational operators, 4-10
Repeat specifications, 12-24/25
Required OPEN statement specifiers, 11-28
RETURN statement, 9-27/28
REWIND specifiers
ERR, 11-56
IOSTAT, 11-56
REWIND statement, 11-56
Run-time options, 15-3

S descriptor, 12-14/15
SAVE statement, 7-26/27
Scale factors, 12-22/24
SELECT CASE statement, 6-25/26
Sequential access, 11-3
SEQUENTIAL specifier, 11-48
SHARED specifier, 11-35, 11-48
SHIFT intrinsic function, 9-16
SIGN intrinsic function, 9-13
SIN intrinsic function, 9-18
SINH intrinsic function, 9-18
SKIPFILE specifiers
ERR, 11-53/54
IOSTAT, 11-54
SKIPFILE statement, 11-53/54
Slash descriptor, 12-26
SNGL intrinsic function, 9-13
Source line errors, D-11
Source listing, B-1/4
SP descriptor, 12-14/15
SPACE directive, 2-11
Spacing descriptors
T descriptor, 12-21/22
X descriptor, 12-21
Special characters, 2-7
Specification statements
COMMON, 7-11/13
DIMENSION, 7-2/3
EQUIVALENCE, 7-15/16
explicit CHARACTER, 7-6/7
explicit type, 7-4/6
EXTERNAL, 7-23/24
IMPLICIT, 7-8/9
IMPLICIT NONE, 7-9
INTRINSIC, 7-25/26
PARAMETER, 7-10
SAVE, 7-25/26
SPOOLFILE specifier, 11-35, 11-48
SQRT intrinsic function, 9-13
SS descriptor, 12-14/15
START specifier, 11-35
Statement
field, 2-3
functions, 9-2/3
Statement label field, 2-2
Statement labels
of DO statements, 6-15, 6-21/22
of ELSE IF THEN statements, 6-9/12
of ELSE statements, 6-11/12
of GOTO statements, 6-2/4
of the arithmetic IF, 6-5
Status indicator, 11-61
STATUS specifier, 11-35/36
STOP statement, 6-28
Storage dictionary, B-5/6
Subprograms
and ENTRY, 9-26/27
definition, 2-1, 9-1
dummy arguments, 9-25
mismatched argument list, 9-32
processing arrays, 9-30
processing of arguments, 9-31
referencing, 9-20/21
returns from, 9-27/28
Subroutine
calling conventions, 9-21/22
definition, 9-1
dummy arguments, 9-25
internal, 9-24
referencing, 9-22/23
returns, 9-27/28
subprogram, 9-21/22
SUBROUTINE statement, 9-21/22
Subscript expression, 3-16
Symbolic cross-reference, B-7/8
Symbolic names
definition, 3-13
in COMMON statement, 7-11
in DATA statement, 8-1
in NAMELIST statement, 11-19
in PARAMETER statement, 7-10
System directive lines, 2-7

T descriptor, 12-21/22
TAN intrinsic function, 9-18
TANH intrinsic function, 9-18
Tape files as a data source, 15-5/6
Terminal errors, D-1
Terminal statement of the DO, 6-17

TYPE statement, 11-13, 11-17/18
Type statements
  explicit CHARACTER, 7-6/7
  explicit type, 7-4/6
  IMPLICIT, 7-8/9
  IMPLICIT NONE, 7-9

Unconditional GO TO statement, 6-2
Unformatted
  input/output records, A-1
  READ statement, 11-11/12
  records, 11-1/2
  WRITE statement, 11-16/19
UNFORMATTED specifier, 11-48
UNIT specifier, 11-5, 11-36, 11-48/49
USER directive, 2-11/12
USER specifier, 11-36
Using data from cards, 15-6

Value separator in list-directed
  output, 11-26
Variables
  definition, 3-14
  implicit typing conventions, 3-13/14
VOLUME specifier, 11-37

WAIT specifier, 11-37
Warning errors, D-1
WRITE statement, 11-13, 11-16/18

X compile option, 15-2
X descriptor, 12-21

Z editing
  and repeat specification, 12-24/25
  description, 12-13/14

Gould Inc., Computer Systems Division
6901 W. Sunrise Blvd.
P. O. Box 409148
Fort Lauderdale, FL 33340-9148
Telephone (305) 587-2900

GOULD
Electronics

Users Group Membership Application

USER ORGANIZATION: _____

REPRESENTATIVE(S): _____

_____

_____

ADDRESS: _____

_____

TELEX NUMBER: _____ PHONE NUMBER: _____

NUMBER AND TYPE OF GOULD CSD COMPUTERS: _____

_____

_____

_____

APPLICATIONS (Please Indicate)

1. EDP
   - A. Inventory Control
   - B. Engineering & Production Data Control
   - C. Large Machine Off-Load
   - D. Remote Batch Terminal
   - E. Other

2. Communications
   - A. Telephone System Monitoring
   - B. Front End Processors
   - C. Message Switching
   - D. Other

3. Design & Drafting
   - A. Electrical
   - B. Mechanical
   - C. Architectural
   - D. Cartography
   - E. Image Processing
   - F. Other

4. Industrial Automation
   - A. Continuous Process Control Op.
   - B. Production Scheduling & Control
   - C. Process Planning
   - D. Numerical Control
   - E. Other

5. Laboratory and Computational
   - A. Seismic
   - B. Scientific Calculation
   - C. Experiment Monitoring
   - D. Mathematical Modeling
   - E. Signal Processing
   - F. Other

6. Energy Monitoring & Control
   - A. Power Generation
   - B. Power Distribution
   - C. Environmental Control
   - D. Meter Monitoring
   - E. Other

7. Simulation
   - A. Flight Simulators
   - B. Power Plant Simulators
   - C. Electronic Warfare
   - D. Other

8. Other

Please return to:

Users Group Administrator

Date: _____

Gould Inc., Computer Systems Division Users Group. . .

The purpose of the Gould CSD Users Group is to help create better User/User and User/Gould CSD Communications.

There is no fee to join the Users Group. Simply complete the Membership Application on the reverse side and mail to the Users Group Administrator. You will automatically receive Users Group Newsletters, Referral Guide and other pertinent Users Group Activity information.

Fold and Staple for Mailing

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 947 FT. LAUDERDALE, FLORIDA

POSTAGE WILL BE PAID BY ADDRESSEE

GOULD INC., COMPUTER SYSTEMS DIVISION
ATTENTION: USERS GROUP ADMINISTRATOR
6901 W. SUNRISE BLVD.
P. O. BOX 409148
FT. LAUDERDALE FL 33340-9970

(Detach Here)

Fold and Staple for Mailing

GOULD
Electronics