
**MASS STORAGE FORTRAN
VERSION 3A/B
REFERENCE MANUAL**

**CONTROL DATA[®]
MASS STORAGE OPERATING SYSTEM
INTERACTIVE TERMINAL-ORIENTED SYSTEM**

REVISION RECORD

REVISION	DESCRIPTION
A (5/72)	Original printing of Mass Storage FORTRAN Version 3.0 for MSOS Version 4.0.
B (1/73)	This printing includes double precision floating point package.
C (7/74)	Revision for MSOS 4
D (2/75)	This revision includes a glossary and PSR changes 2044, 2192, 2375, 2725, and 2837.
E (5/76)	This revision is for FORTRAN Version 3.3 and PSR changes 3552, 3328, 3469, 3335, 3510, 3868, and 3934. See List of Effective Pages for pages affected in this revision.
F (9/76)	Manual revised for MSOS Version 5. See List of Effective Pages for pages affected in this revision.
G (10/78)	Manual revised to reflect ITOS 2.0 release. The front cover, title page, pages iii/v through xii, and Index-1, Index-3, Index-4, and Index-5 have been revised. Page P-1 has been added.
H (7/80)	Manual revised to incorporate PSR FT3A035. Refer to the List of Effective Pages for pages affected by this revision.
J (7/81)	Manual revised to clarify runtime values.
Publication No. 60362000	

REVISION LETTERS I, O, Q, S, X, AND Z ARE NOT USED

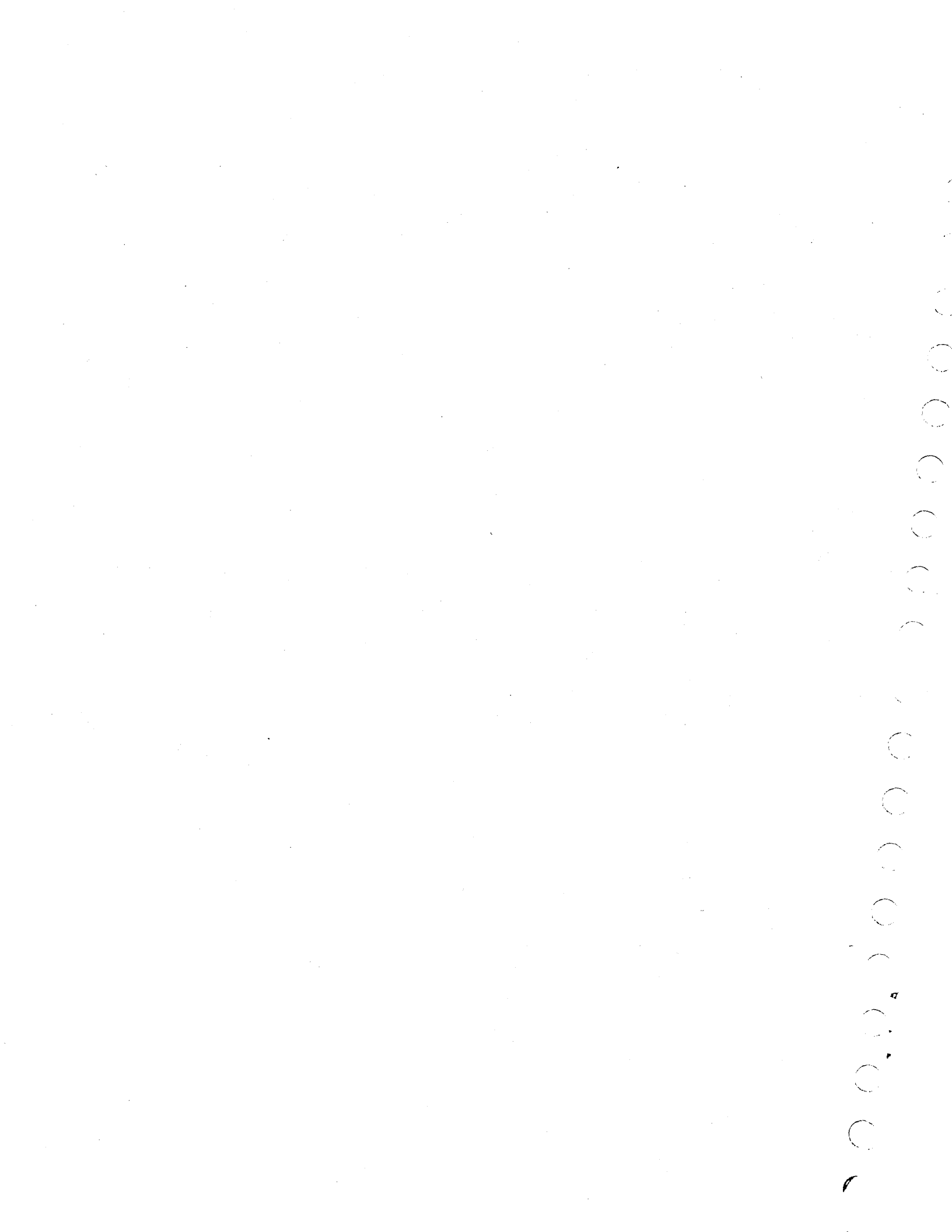
Address comments concerning this manual to:
Control Data Corporation
Publications and Graphics Division
4455 Eastgate Mall
La Jolla, California 92037
 or use Comment Sheet in the back of this manual.

© 1972, 1973, 1974, 1975, 1976, 1978, 1980, 1981
 by Control Data Corporation
 Printed in the United States of America

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual, are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

PAGE	REV	PAGE	REV	PAGE	REV	PAGE	REV	PAGE	REV
Cover	--	5-36	E	9-22	E	Index-6	E		
Title Page	--	5-37	C	9-23	F	Comment			
ii	J	6-1	C	9-24	H	Sheet	J		
iii/iv	J	6-2	H	9-25	E	Cover	--		
v/vi	G	6-3	C	9-26	E				
vii	G	6-4	F	9-27	H				
viii	H	6-5	C	9-28 thru 9-33	E				
ix thru xi/xii	G	6-6 thru 6-8	E	10-1	E				
1-1	F	6-9	H	10-2	H				
1-2	E	6-10	H	Glossary-1	F				
2-1	F	6-11	E	Glossary-2	F				
2-2	C	6-12	E	Glossary-3	D				
2-3	F	6-13	H	Glossary-4	H				
2-4	E	6-14	E	A-1	E				
2-5	E	6-15	H	A-2	C				
2-6	F	6-16 thru 6-20	E	B-1	D				
2-7	J	6-21 thru 6-27	C	C-1	J				
2-8	E	7-1	F	C-2	E				
2-9	E	7-2	C	C-3	C				
3-1	C	7-3	E	C-4 thru C-16	E				
3-2	C	7-4	C	D-1	J				
3-3	E	7-5	F	D-2 thru D-15	E				
3-4	F	7-6	C	E-1	E				
3-5	C	7-7	D	E-2	E				
4-1	E	7-8	C	F-1	F				
4-2	E	7-9	D	G-1	C				
5-1 thru 5-6	C	7-10	E	G-2	E				
5-7	E	7-11	C	G-3	C				
5-8	D	7-12	E	G-4	C				
5-9	C	7-13 thru 7-15	C	H-1	F				
5-10	E	7-16	E	I-1	E				
5-11	C	7-17	H	J-1	F				
5-12	E	7-18	C	K-1	C				
5-13	E	8-1	F	K-2	C				
5-14	C	8-2	F	L-1	E				
5-15	E	8-3 thru 8-11	E	L-2	D				
5-16	E	8-12	F	L-3	F				
5-17	C	8-13	F	L-4	E				
5-18	F	9-1	F	L-5	E				
5-19	E	9-2 thru 9-7	C	M-1	F				
5-20	C	9-8 thru 9-10	D	M-2	C				
5-21	H	9-11	E	M-3	C				
5-22	C	9-12	E	N-1/N-2	H				
5-23	F	9-13	F	O-1	E				
5-24	E	9-14	C	P-1	G				
5-25	F	9-15 thru 9-17	H	Index-1	G				
5-26	C	9-18	D	Index-2	H				
5-27 thru 5-29	F	9-19	C	Index-3	H				
5-30 thru 5-32	D	9-20	D	Index-4	G				
5-33 thru 5-35	H	9-21	D	Index-5	H				



PREFACE

This publication describes the external features of the Mass Storage FORTRAN Version 3 language and the information necessary to produce Mass Storage FORTRAN programs for the CONTROL DATA® 1704/1714/1774/1784 computers and CYBER 18 computers.

It is assumed that the reader has some knowledge of an existing FORTRAN language and the Mass Storage Operating System (MSOS).

Mass Storage FORTRAN, with the use of a compile time option, is a subset of ANSI X3.9-1966 American National Standard FORTRAN. All FORTRAN source decks written according to the guidelines provided by this document are compiled according to ANSI standards.

Mass Storage FORTRAN operates under Mass Storage Operating Systems 4.3 and 5 and Interactive Terminal-Oriented System (ITOS) 2.0.

Related manuals in which the FORTRAN user may find additional information are:

<u>Description</u>	<u>Publication No.</u>
MSOS 5 Reference Manual	96769400
Macro Assembler Reference Manual	60361900
1700 Computer System Codes	60163500
Small Computer Maintenance Monitor Reference Manual	39520200
MSOS 5 Instant	96769530
File Manager Version 1 Reference Manual	39520600
MSOS 5 Installation Handbook	96769410
Small Computer Maintenance Monitor Instant	39521700
Interactive Terminal-Oriented System (ITOS) Version 2 Reference Manual	96769240
Interactive Terminal-Oriented System (ITOS) Version 2 Installation Handbook	60475200

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features and parameters.



CONTENTS

1	INTRODUCTION	1-1
	1.1 Product Elements	1-1
	1.2 Product Configurations	1-1
	1.3 Product Hardware Requirements	1-2
2	DATA FORMAT	2-1
	2.1 Data Elements	2-1
	2.2 Data Types	2-1
	2.2.1 Integer Type Data	2-3
	2.2.2 Single Type Data	2-5
	2.2.3 Byte Type Data	2-5
	2.2.4 Signed Byte Type Data	2-5
	2.2.5 Real Type Data	2-6
	2.2.6 Double Precision Type Data	2-7
	2.3 Symbolic Names	2-8
	2.4 Data Names	2-8
3	EXPRESSIONS	3-1
	3.1 Arithmetic Expressions	3-1
	3.1.1 Rules for Forming Arithmetic Expressions	3-1
	3.1.2 Order of Evaluation	3-2
	3.1.3 Mixed Mode	3-3
	3.2 Relational Expression	3-3
	3.3 Logical Expression	3-4
	3.3.1 Formation of Logical Expression	3-4
	3.3.2 Order of Evaluation	3-5
4	STATEMENTS	4-1
	4.1 Classification	4-1
	4.1.1 Executable Statements	4-1
	4.1.2 Nonexecutable Statements	4-1
	4.2 Statement Format	4-1

5	EXECUTABLE STATEMENTS	5-1
5.1	Assignment Statements	5-1
5.1.1	Arithmetic Assignment Statement	5-1
5.1.2	Label Assignment Statement	5-2
5.2	Control Statements	5-3
5.2.1	GO TO Statements	5-4
5.2.2	Arithmetic IF Statement	5-7
5.2.3	Logical IF Statement	5-7
5.2.4	CALL and RETURN Statement	5-8
5.2.5	CONTINUE Statement	5-8
5.2.6	Program Control Statements	5-8
5.2.7	DO Statement	5-9
5.3	I/O Statements	5-18
5.3.1	I/O Devices	5-18
5.3.2	Mass Storage Files	5-18
5.3.3	OPEN Statement	5-18
5.3.4	READ and WRITE Statements	5-20
5.3.5	Auxiliary I/O Statements	5-28
5.3.6	Tape Records and Blocks	5-30
5.3.7	Mass Storage Records and Sectors	5-32
5.3.8	Printing of Carriage Control	5-33
5.4	ASSEM Statement	5-34
6	NONEXECUTABLE STATEMENTS	6-1
6.1	Specification Statements	6-1
6.1.1	DIMENSION Statement	6-1
6.1.2	COMMON Statement	6-2
6.1.3	EQUIVALENCE Statement	6-4
6.1.4	Type Statements	6-6
6.1.5	Byte Statements	6-7
6.2	DATA Statement	6-9
6.3	FORMAT Statement	6-11
6.3.1	Field Descriptors	6-12
6.3.2	Field Separators	6-12
6.3.3	Numeric Conversion	6-13
6.3.4	Alphanumeric Conversion	6-20
6.3.5	Editing Specifications	6-21
6.3.6	New Record Specifications	6-23
6.3.7	Blank Field Specification	6-24
6.3.8	Repeated Format Specifications	6-25
6.3.9	Format Specification in Arrays	6-27

7	PROCEDURES AND SUBPROGRAMS	7-1
	7.1 Arguments	7-3
	7.1.1 Actual	7-3
	7.1.2 Dummy	7-3
	7.2 Statement Function	7-3
	7.3 Supplied Functions	7-5
	7.3.1 Intrinsic Function	7-7
	7.3.2 Basic External Function	7-7
	7.4 Subprograms	7-9
	7.4.1 Function Subprogram	7-10
	7.4.2 SUBROUTINE Subprogram	7-12
	7.4.3 EXTERNAL Statement	7-13
	7.4.4 RELATIVE Statement	7-16
	7.4.5 CALL Statement	7-16
	7.4.6 RETURN Statement	7-17
	7.4.7 Block Data Subprogram	7-17
8	COMPILATION AND EXECUTION	8-1
	8.1 Compilation	8-1
	8.2 Execution	8-12
	8.3 Program Operating Procedures	8-12
9	FORTRAN MULTIPROGRAMMING	9-1
	9.1 Re-entrant FORTRAN	9-1
	9.1.1 Priorities	9-1
	9.1.2 Re-entrancy	9-2
	9.1.3 FORTRAN Library	9-2
	9.1.4 FORTRAN READ/WRITE Statement Processor	9-3
	9.1.5 FORTRAN/Monitor Run-Time Interface (FORTRA)	9-3
	9.1.6 Encode/Decode	9-3
	9.1.7 Run-Anywhere Programs	9-4
	9.2 Format Specifications	9-4
	9.2.1 FORMAT Statement	9-4
	9.2.2 Format Conversion	9-5
	9.2.3 Conversion Specifications	9-5
	9.2.4 Editing Specifications	9-10
	9.2.5 Special Character Specifications	9-11
	9.2.6 Repeated Format Specifications	9-12
	9.3 FORTRAN READ/WRITE Statement Processor	9-12
	9.3.1 WRITE Statement	9-12
	9.3.2 READ Statement	9-12
	9.3.3 Statement Processor	9-13
	9.3.4 CALL SETBFR	9-13

9.3.5	Restrictions	9-13
9.3.6	Format Errors	9-15
9.3.7	I/O Errors	9-15
9.4	ENCODE/DECODE Calls	9-15
9.4.1	ENCODE	9-16
9.4.2	ENCODE Macro	9-17
9.4.3	DECODE	9-17
9.4.4	DECODE Macro	9-19
9.4.5	ENCODE/DECODE Error Detection	9-19
9.4.6	Additional Formatting Routines	9-20
9.5	FORTTRAN/Monitor Run-Time Package	9-23
9.5.1	READ/WRITE Calling Sequence	9-24
9.5.2	Scheduler and Timer	9-28
9.5.3	Miscellaneous Calling Sequences	9-29
9.5.4	Buffered Input/Output	9-31
10	MACRO FACILITY	10-1
10.1	Macro Definition	10-1
10.2	Macro Call	10-1
10.3	Restrictions	10-2
10.4	Limits	10-2
Glossary		Glossary-1
Appendix A	Communication Between FORTRAN and Assembly Language Programs	A-1
Appendix B	Table Capacities	B-1
Appendix C	Single-Precision Floating-Point Package	C-1
Appendix D	Double-Precision Floating-Point Package	D-1
Appendix E	Coding Hints	E-1
Appendix F	Hardware Requirements	F-1
Appendix G	Arrays	G-1
Appendix H	ASCII Codes	H-1
Appendix I	Load Map Comment Feature	I-1
Appendix J	Optimizations	J-1
Appendix K	FORTTRAN Character Set	K-1
Appendix L	FORTTRAN Compilation Errors	L-1
Appendix M	FORTTRAN Execution Errors	M-1
Appendix N	FORTTRAN Miscellaneous Errors	N-1
Appendix O	MS FORTRAN Reserved Word List	O-1
Appendix P	FORTTRAN Operation in ITOS User Area	P-1
Index		Index-1

FIGURES

Figure

2-1	Data Types and Subdivisions	2-2
6-1	Common Block	6-3
G-1	Array Successor Function $a + A * (b-1) + A * B * (c-1)$	G-3

TABLES

Table

5-1	DO Statement Parameters	5-11
7-1	Subdivision of Procedures and Subprograms	7-2
7-2	Permissible Arguments for Functions and Subprograms	7-4
7-3	Intrinsic Functions	7-6
7-4	Basic External Functions	7-8
7-5	Basic External Functions, Predetermined Results	7-9
7-6	Differences Between Function and Subroutine Subprograms	7-10
G-1	Value of a Subscript	G-2



INTRODUCTION

1

1.1 PRODUCT ELEMENTS

The Mass Storage FORTRAN Version 3 product is composed of five basic elements:

- A Variant FORTRAN Compiler — This compiler version has a larger number of overlays; the largest overlay is approximately 8K. It requires more mass memory than the B variant and is slower in compilation speed.
- B Variant FORTRAN Compiler — This compiler has fewer overlays than the A variant; the largest overlay is approximately 16K. This variant is faster than the A variant. Both compilers process source statements identically and generate similar object codes.
- Re-entrant ENCODE/DECODE Run-time — This run-time library runs in the foreground and has the characteristics for multiprogramming described in Chapter 9.
- Non-Re-entrant ENCODE/DECODE Run-time — This run-time library runs in the background and has identical user interface as the re-entrant ENCODE/DECODE run-time library as described in Chapter 9. This run-time library is designed for use in debugging programs to run in the foreground.
- FORTRAN I/O Run-time — This run-time library runs in the background and has the capability described in this manual (except Chapter 9). In general, it has more extensive capability than the other two run-times.

1.2 PRODUCT CONFIGURATIONS

Several product configurations are possible using the five elements of the product.

Only one variant of the compiler may be present in a given MSOS system. With the selected compiler, the re-entrant ENCODE/DECODE run-time may be used (must be core-resident). Either the non-re-entrant ENCODE/DECODE or FORTRAN I/O run-times may be in the background. In addition, if the FORTRAN I/O run-time is in the background, the non-duplicative functions present in the non-re-entrant ENCODE/DECODE can also be in the background.

1.3 PRODUCT HARDWARE REQUIREMENTS

The MSOS reference manual should be consulted for the specific hardware options which are available.

The minimum system memory requirements for MSOS do not include any of the elements of Mass Storage FORTRAN. If the A variant of the compiler is used, the minimum memory requirement is 24K. The B variant minimum is 32K. The foreground ENCODE/DECODE run-time requires an additional 4K of memory for single-precision floating-point or 8K for double-precision floating-point.

2.1 DATA ELEMENTS

A data element is a single-valued unit of data which may be uniquely referenced. It may be any of the six types outlined in the following section. A data element may occupy part of a word (byte), a full word (integer or single), two full words (real), or three full words (double precision). The value of a data element may be altered during program execution.

The following expression contains six data elements.



2.2 DATA TYPES

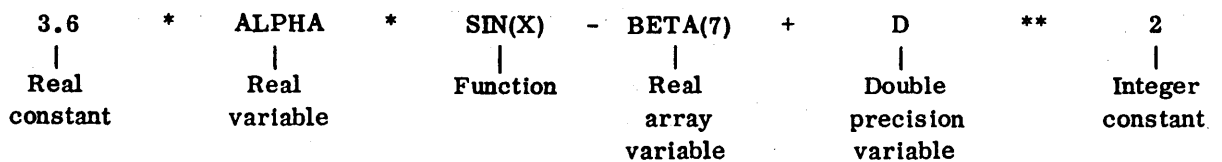
MS FORTRAN recognizes six types of data:

- Integer
- Single
- Real
- Double precision
- Byte
- Signed byte

Operations with data elements must take into account their type, since each has its own mathematical significance and word structure.

The type of a data element is indicated either by the first letter of its symbolic name or by a specification statement. Data types are shown in Figure 2-1.

Based on the six data elements in the preceding example, the data types are:



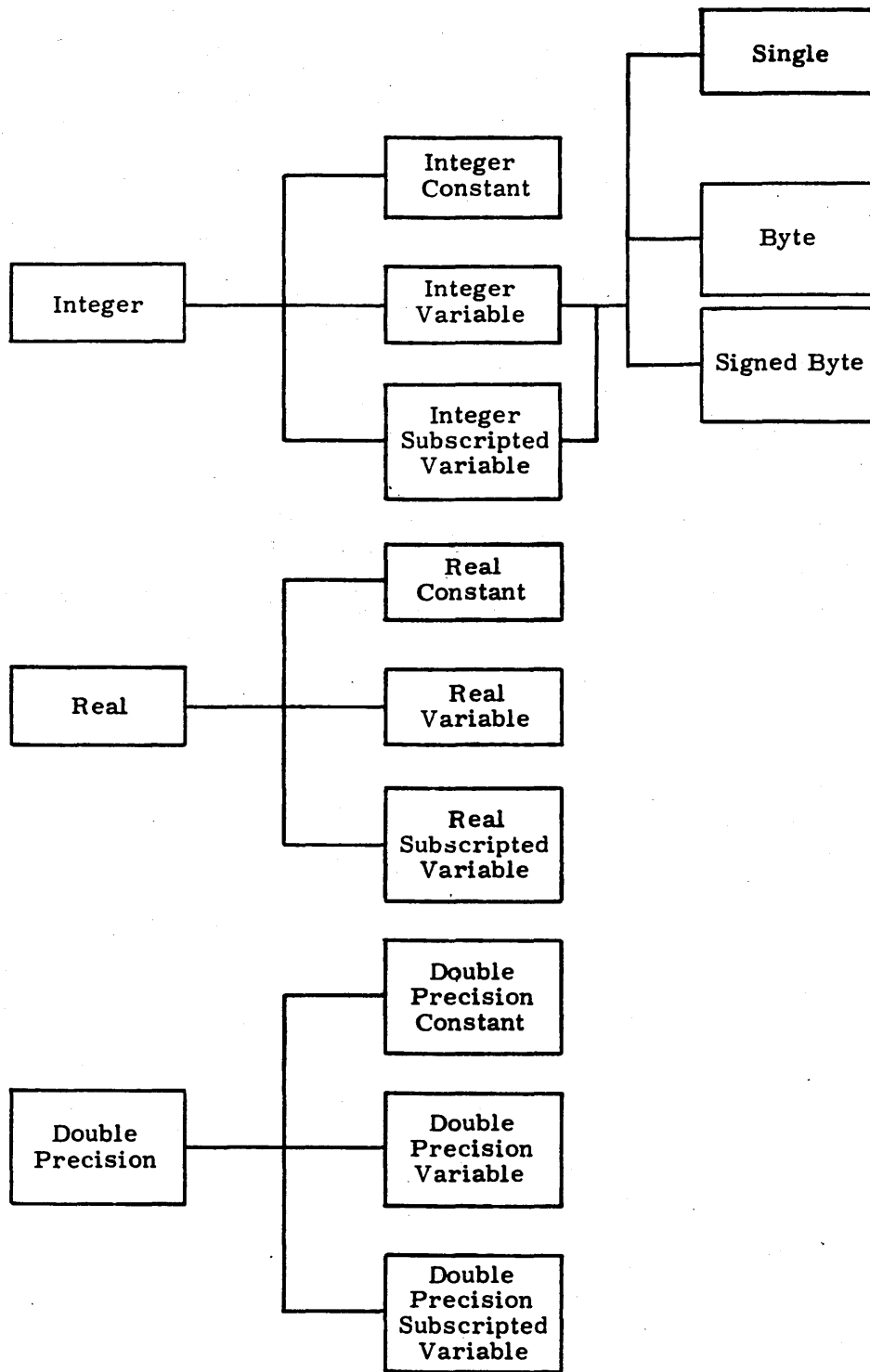
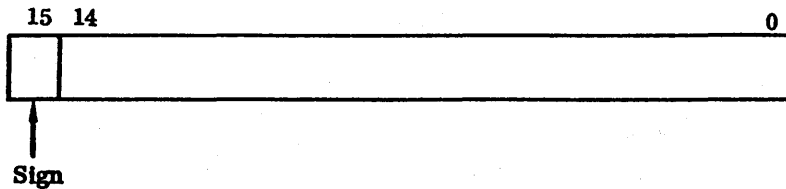


Figure 2-1. Data Types and Subdivisions

2.2.1 INTEGER TYPE DATA

An integer is a whole number expressed without a decimal point. It may be used as a subscript, an exponent, or in calculations that do not involve fractional parts. An integer occupies 16 bits of storage, or one computer word. The most significant bit is the sign bit.



The range of integer in magnitude is $0 \leq |n| \leq 2^{15} - 1$.

There are three integer types:

- **Constant** The value of an integer constant is stated explicitly in an expression. In MS FORTRAN, integer constants may be
 - Decimal** Decimal integer constants consist of one to five decimal digits. If the range of $0 \leq |n| \leq 2^{15} - 1$ is exceeded, a diagnostic is provided. Leading zeros are ignored.
 - Hexadecimal** Hexadecimal integer constants are distinguished from decimal integer constants by a \$ sign immediately preceding the string of digits. Hexadecimal integer constants consist of one to four hexadecimal digits. If this maximum is exceeded, the constant is treated as zero and a compiler diagnostic is provided. Leading zeros are ignored.
 - Octal** An octal integer constant consists of one to five octal digits. Its use is restricted to PAUSE_n and STOP_n statements in which n is an octal constant.

Equivalent decimal, hexadecimal, and octal integers are:

<u>Decimal Integer</u>	<u>Hexadecimal Equivalent</u>	<u>Octal Equivalent</u>
123	7B	173
239	EF	357
8405	20D5	20325

Hollerith

A Hollerith integer constant has the form nHf or nRf

Where: n is the unsigned decimal integer representing number of characters in string. Must be greater than zero and not more than 2 when used in an expression.

f is the string of characters.

H is left justified with blank fill.

R is right justified with zero fill.

Examples:

<u>Hollerith Constant</u>	<u>Hexadecimal Equivalent</u>
2HAB	4142
2HA	4120
1RA	0041
2RAB	4142

- **Variable** An integer identified by a symbolic name (Section 2.4) and capable of assuming a range of values during program execution is an integer variable. It may be designated a simple integer variable to distinguish it from an integer subscripted variable.

SINGLE, BYTE, and SIGNED BYTE data types are subsets of the integer variable.

- **Subscripted Variable** This type of integer is a symbolic name with one, two, or three associated subscripts enclosed in parentheses. It is used to reference elements in an array (Appendix G) of successive memory locations. The name is typed integer by alphanumeric format (Section 2.4) or by declaration (Section 6.1.4). The subscripts must be integer constants, integer variables, or integer expressions. Permissible forms of subscripts are

<u>Form</u>	<u>Example</u>
(i)	(I)
(c)	(3)
(i±d)	(I+5)
(c*i)	(3*I)
(c*i±d)	(3*I+5)
i	Integer variable
c } d }	Integer constants
*	Arithmetic operator; multiplication
+	Arithmetic operator; addition or positive value
-	Arithmetic operator; subtraction or negative value

Before an array can be used in a program, its name and dimensions must be declared in a DIMENSION, COMMON, or TYPE statement (Section 6.1.4). When so declared, the subscripts are the actual dimensions of the array.

2.2.2 SINGLE TYPE DATA

This data type is the same as an integer variable. Dimension information may be given. When the ANSI option has been declared, appearance of a name in a SINGLE statement declares that each data element specified occupies a single storage unit.

2.2.3 BYTE TYPE DATA

A byte is an integer part (16 bits or less) of an integer variable. It is unsigned and may assume positive and zero values. To assume negative values the byte must be a full integer word (16 bits).

NOTE

When byte or signed byte integer parts of integer variables are used in subprogram parameter strings, the address of the integer variable is passed to the subprogram (not a redefined integer part address as defined by the BYTE or SIGNED BYTE declaration). The subprogram will then obtain the complete integer variable value when the byte parameter is referenced.

2.2.4 SIGNED BYTE TYPE DATA

A signed byte of an integer word may assume positive, negative, and zero values. In the special case where a signed byte is one bit, it has the value +0 or -0.

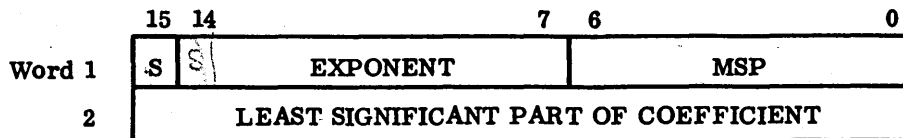
NOTE

When byte or signed byte integer parts of integer variables are used in subprogram parameter strings, the address of the integer variable is passed to the subprogram (not a redefined integer part address as defined by the BYTE or SIGNED BYTE declaration). The subprogram will then obtain the complete integer variable value when the byte parameter is referenced.

2.2.5 REAL TYPE DATA

A real data element can have a fractional part as well as an integer part and is always expressed with a decimal point. It is used in calculations that require decimal approximations.

A real number occupies 32 bits or two words.



Where: S is the sign bit

MSP is the most significant part of the coefficient

The approximate range of a real number is $10^{-39} < |n| < 10^{39}$. Precision is approximately seven decimal digits. (Refer to Appendix C.)

There are three real types:

- Constant** The value of a real constant data element is expressed by an integer part, a decimal point, and a fractional part, in that order. It may be followed by the letter E and an optionally signed exponent representing a power of ten. In the following examples, n is the integer part, d the decimal (fractional) part, and s the exponent representing a power of 10. (Refer to the Constant description for double precision type, page 2-7.)

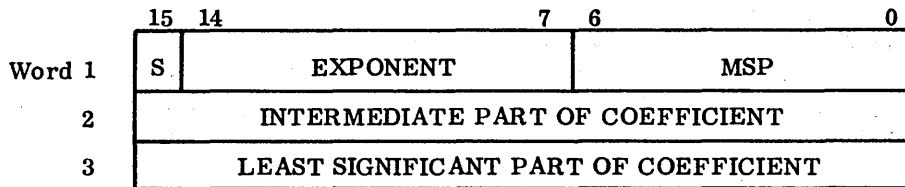
<u>Form</u>	<u>Example</u>
n.d	345.67
.d	.34567
n.	34567.
.dE±s	.34567E+5
n.E±s	34567.E-05
n.dE±s	345.67E-03
- Variable** A real variable data element is identified by a symbolic name (Section 2.3). It is capable of assuming a range of values during program execution. A real variable is designated a simple real variable to distinguish it from a real subscripted variable.
- Subscripted Variable** A real subscripted variable is a symbolic name with one, two, or three subscripts enclosed in parentheses. It is used to reference the elements in an array of memory locations. The name is typed real by alphanumeric format (Section 2.4) or by declaration (Section 6.1.4). The subscripts must be integer (constants, variables, or expressions). Permissible forms of subscripts are

<u>Form</u>	<u>Example</u>
(I)	(I)
(c)	(3)
(I±d)	(I+5)
(c*i)	(3*I)
(c*i±d)	(3*I+5)
i	Integer variable
c } d }	Integer constants
*	Arithmetic operator: multiplication
+	Arithmetic operator: addition or positive value
-	Arithmetic operator: subtraction or negative value

2.2.6 DOUBLE PRECISION TYPE DATA

A double precision data element can have a fractional part as well as an integer part and is always expressed with a decimal point. It is used in calculations that require decimal approximations of more accuracy than that obtainable with the use of a single precision data element.

A double precision number occupies 48 bits or three words.



Where: S is the sign bit

MSP is the most significant part of the coefficient

The approximate range of a double-precision number is $10^{-39} < |n| < 10^{39}$. Precision is approximately 11.5 decimal digits in the software version, and approximately 9.5 digits in the firmware version runtime.

The double precision types are:

- Constant The value of a double-precision constant data element is expressed by an integer part, a decimal point, and a fractional part followed by the letter D and an optionally signed exponent representing a power of ten. A constant

with a decimal point, but without an exponent, is classed as a real constant. In the following examples, n is the integer part, d the decimal (fractional) part, and s the exponent representing a power of 10.

<u>Form</u>	<u>Example</u>	
n.dD±s	345.67D-03	
.dD±s	.34567D+5	
n.D±s	34567.D-05	
n.d	838.8607	(real)
n.d	838.8608	(real)
.d	.08388607	(real)
.d	.08388608	(real)
n.	-8388607.	(real)
n.	-8388607.0	(real)

- **Variable** A double-precision variable data element is identified by a symbolic name (Section 2.3). It is capable of assuming a range of values during program execution. A double-precision variable is designated a simple double-precision variable to distinguish it from a double-precision subscripted variable.
- **Subscripted Variable** A double-precision subscripted variable is a symbolic name with one, two, or three subscripts enclosed in parentheses. It is used to reference the elements in an array of memory locations. The name is typed by declaration (Section 6.1.4). The subscripts must be integer constants, variables, or expressions. See Section 2.2.5 for permissible forms of subscripts.

2.3 SYMBOLIC NAMES

Both type of symbolic names consist of one to six alphanumeric characters, the first of which must be alphabetic:

- **Data names** — Reference simple variables, arrays, the elements of an array, bytes, and data blocks
- **Procedure names** — Reference statement functions, intrinsic functions, external functions, subroutines, and certain external procedures

2.4 DATA NAMES

A data name identifies any of the variable data elements described in this section. It also references a data block. In the absence of an explicit declaration, type is implied by the first letter of the name: I, J, K, L, M, and N imply type integer†; any other letter implies type real.

† Byte and signed byte are always considered integer variable.

Example:

<u>Integer Variable</u>	<u>Real Variable</u>
IOTA	A65302
MATRIX	BETA
J	C
K2S04	ALPHA (7)



An expression is a set of data elements combined by operators and parentheses to produce, upon execution, a single-valued result. An expression can be a single data element, a constant, or a variable, or it can be a complex string of data elements and operators nested with parentheses.

There are three kinds of expressions: arithmetic, relational, and logical, and each has its own operators.

3.1 ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of arithmetic operators and data elements which, when evaluated by execution, produce a single numerical value. Both the expression and its data elements identify integer, real, or double-precision values. Byte and signed byte are synonymous with type integer.

Arithmetic operators

- + Addition or positive value
- Subtraction or negative value
- * Multiplication
- / Division
- ** Exponentiation

Arithmetic data elements

- Constants
- Simple or subscripted variables
- Function references (refer to Chapter 7)

3.1.1 RULES FOR FORMING ARITHMETIC EXPRESSIONS

Consecutive arithmetic operators are not allowed in an expression. If a minus sign is used to indicate a negative data element, the sign and the element must be enclosed in parentheses if preceded by an operator.

$B * A / (-C)$

$A * (-C)$

As in ordinary mathematical notation, parentheses may be used to indicate grouping, but they may not be used to indicate multiplication.

Any arithmetic data element or expression may be raised to a positive or negative integer element or expression.

M^{**N}
 $(X+Y)^{**I}$
 $(A+B)^{**(-J)}$
 $IVAL^{**(K+2)}$

Only a positive real or double-precision data element or expression can be raised to a real or double-precision power.

$ALPHA^{**3.2}$
 $(X+Y)^{**A}$
 $(A+B)^{**(C+3)}$

Because of truncation, integer expressions cannot be commuted. $I*J/K$ may not yield the same result as $J/K*I$, as the following example shows.

$4*3/2=6$ but $3/2*4=4$

A data element with a zero value may not be raised to a power valued as zero: thus, any expression that becomes 0^{**0} when evaluated is illegal.

All data elements in an arithmetic expression must have mathematically defined values before the expression can be evaluated.

3.1.2 ORDER OF EVALUATION

Evaluation begins with the innermost expression and proceeds outward in parenthetical expressions within parenthetical expressions.

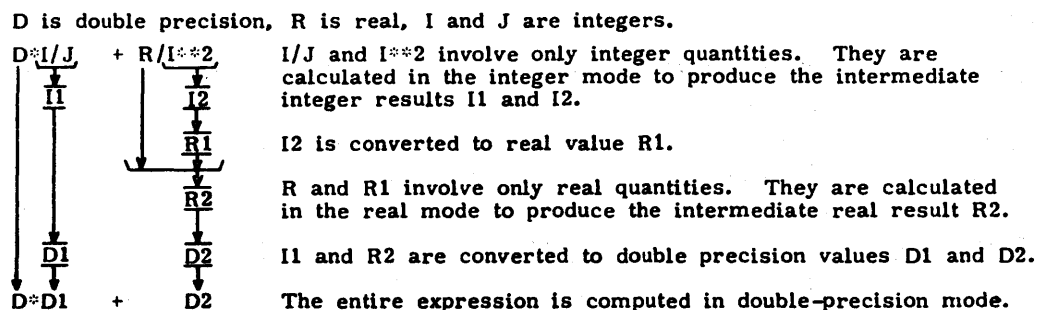
Evaluation proceeds according to the following hierarchy of operators in an expression without parentheses or within a pair of parentheses.

**	Exponentiation	Level 1
/	Division	} Level 2
*	Multiplication	
+	Addition	} Level 3
-	Subtraction	

3.1.3 MIXED MODE

Integer, real, and double-precision quantities may be used in the same arithmetic expression. In such a mixed mode expression, those parts involving purely integer (or real) operations are computed in the integer (or real) mode and the results are converted to real or double-precision. In those parts of the expression involving integer and real quantities, the integer is converted to real; in those parts of the expression involving integer, real, and double-precision quantities, the integer and real are converted to double-precision. Then the entire expression is computed in the real or double-precision mode.

Example:



Example:

For the following statements

```
I = 4*3/2
J = 3/2*4
K = 4.0D0*3/2
```

the results are: I J K
 6 4 4

3.2 RELATIONAL EXPRESSION

Two arithmetic expressions may be combined with a relational operator to form a relational expression. The value of the expression is true or false depending on the relation. A minus zero is always evaluated to be equal to a plus zero.

<u>Relational Operators</u>	<u>Meaning</u>
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to
.LT.	Less than
.LE.	Less than or equal to

Examples:

(A+B).LE.(C+D)

I.EQ.J(K)

(3.*BETA+VALUE).NE.(ALPHA-44.8)

A.GT.16.

In MS FORTRAN, a relational expression is used only within the context of a logical IF statement (Section 5.2.3).

3.3 LOGICAL EXPRESSION

A logical expression is a combination of relational expressions and logical operators such that evaluation of the expression produces a result of true or false.

<u>Logical Operators</u>	<u>Meaning</u>
.NOT.	Logical negation
.AND.	Logical conjunction
.OR.	Logical disjunction

In MS FORTRAN, a logical expression is used only within the context of a logical IF statement (Section 5.2.3). Logical variables are not allowed in MS FORTRAN.

3.3.1 FORMATION OF LOGICAL EXPRESSION

If RE1 and RE2 are relational expressions, the logical operators can be defined as follows:

.NOT.RE1	False only if RE1 is true
RE1.AND.RE2	True only if RE1 and RE2 are both true
RE1.OR.RE2	False only if RE1 and RE2 are both false

.NOT. may appear only in the following combinations with .AND. or with .OR.:

RE1.AND..NOT.RE2

RE1.OR..NOT.RE2

Examples:

A.LE.B.AND.C.EQ.D

F.GT.16..OR.G.GE.3.14

ALPHA.LE.BETA.AND..NOT.GAMMA.EQ.BETA

.NOT.(A.NE.B) which is the same as A.EQ.B

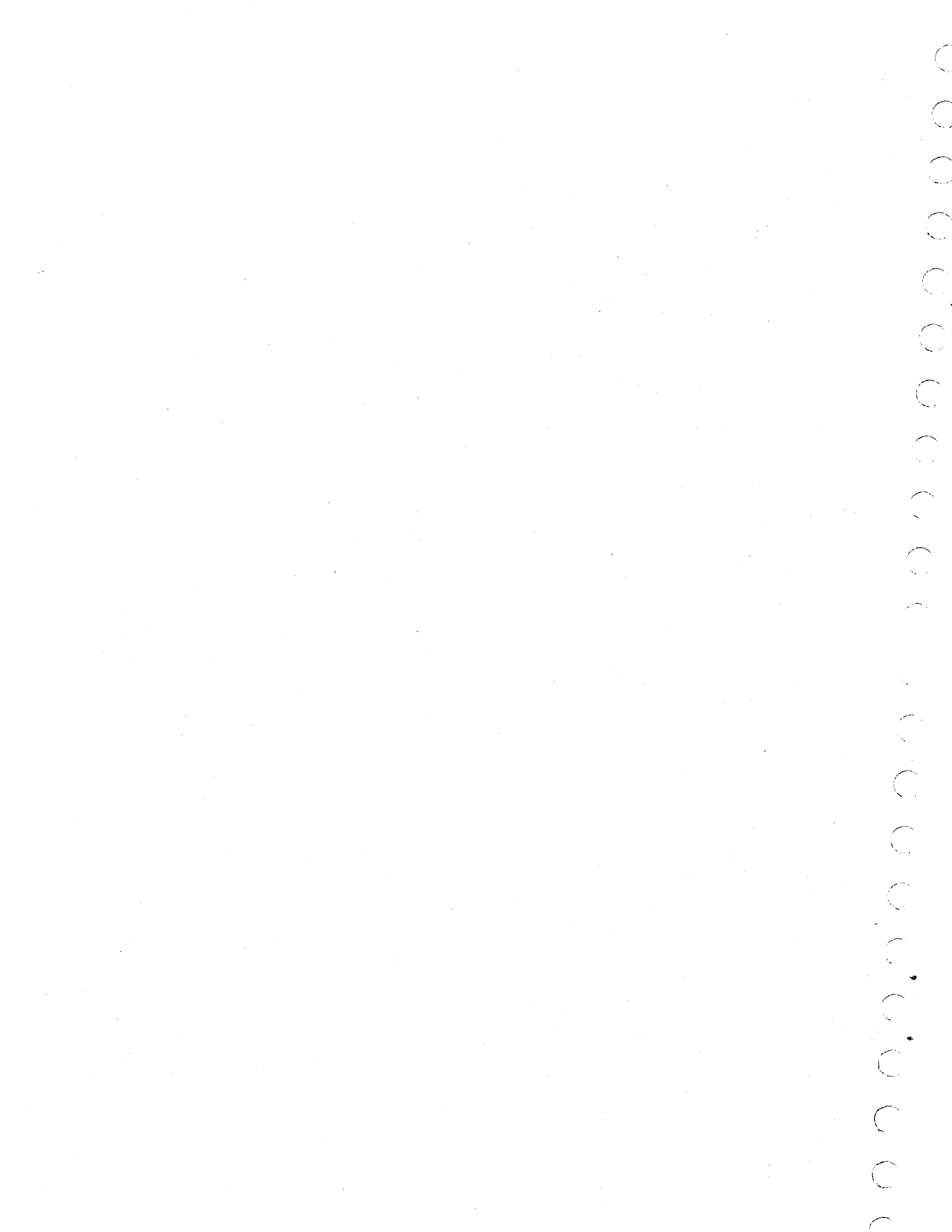
3.3.2 ORDER OF EVALUATION

Within a logical expression, operators are evaluated in the following order:

.NOT.

.AND.

.OR.



4.1 CLASSIFICATION

Statements are the basic steps in a FORTRAN program. In general, statements are executable or nonexecutable.

4.1.1 EXECUTABLE STATEMENTS

Executable statements perform calculations, direct control of the program, and transfer data. Types of executable statements are

Assignment

Control

I/O

4.1.2 NONEXECUTABLE STATEMENTS

Nonexecutable statements provide the compiler with information regarding data structure and storage. Nonexecutable statements are

Specification

Data initialization

Format

Function defining

Subprogram

4.2 STATEMENT FORMAT

Statements are written in 72-column lines. A statement begins on the initial line and may be continued to additional lines. Up to five continuation lines are permitted per statement.

The use of the 72 columns is the same for punched card and paper tape input; however, for paper tape, column 72 indicates a carriage return which serves as a field separator marking the end of an input line.

Blanks may be used freely to improve the appearance of the program, subject to the restrictions on continuation lines.

In writing statements, the columns are used as follows:

<u>Column</u>	<u>Description</u>	<u>Use</u>
1	C } * } \$ }	Comment line. Does not affect program
	+	Comment card. Is printed on the same line as previous statement card (allows statement and comment on same line)
	.	Page eject
	N	Where $1 \leq N \leq 9$ skips N blank lines if there are no other characters on the card. If there are other characters, the card is considered a normal statement.
	M	Macro call (see Macro Facility in section 10)
1 through 5	Statement label	If a statement is to be referenced in a program (such as in control transfer), it is given a number from 1 to 32,767 as a statement label. Otherwise, these columns are blank.
6	Continuation indicator	Where a statement extends beyond one line, additional lines are flagged as continuation lines by placing a character other than zero or blank in column 6. When column 6 is used, the line must contain some useful information or the compiler assumes the programmer made an error.
7 through 71	Statement field	The FORTRAN statement is written in columns 7 through 72 for punched card input and in columns 7 through 71 for paper tape input.
72	Carriage return	A carriage return symbol is placed in column 72 for every line of paper tape input to indicate the end of line.
73 through 80	User application	The programmer uses these columns to sequence source cards; the compiler ignores these columns.

An executable statement causes the program to perform an action such as the assignment of a value, the transfer of control, or the transfer of data. Executable statements are

Assignment

Control

I/O

5.1 ASSIGNMENT STATEMENTS

An assignment statement gives a variable numerical value. The value may be the result of calculation, or it may be assigned by the programmer. Assignment statements may be

Arithmetic assignment

Label assignment

5.1.1 ARITHMETIC ASSIGNMENT STATEMENT

An arithmetic assignment statement assigns a value of a constant, expression, or variable to another variable.

The format is

$$v = e$$

- Where:
- v** is the simple or subscripted variable
 - =** is the assignment symbol which directs the program to compute the value of the expression on the right and place that value in the storage location designated by the variable on the left
 - e** is the arithmetic expression

Examples:

$$I = I + 1$$

$$ALPHA = BETA * DELTA + SIN (X)$$

$$JOTA (K) = IVAL * *2 + IFOX (Y)$$

If the arithmetic assignment statement involves mixed mode, the data type of e may be converted according to the rules:

<u>If v type is:</u>	<u>And e type is:</u>	<u>The assignment rule is:</u>
Integer	Integer	Assign
Integer	Real	Fix and assign
Integer	Double precision	Dfix and assign
Real	Integer	Float and assign
Real	Real	Assign
Real	Double precision	Single and assign
Double precision	Integer	Dflt and assign
Double precision	Real	Double and assign
Double precision	Double precision	Assign

Definition of Rules

Assign	Transmit the value, without change, to v.
Fix/Dfix	Truncate any fractional part of the value and transform that result to the form of an integer.
Float	Transform the value to the form of a real number.
Dflt	Transform the value to the form of a double-precision number.
Single	Truncate the value to form a real number.
Double	Express the value in the form of a double-precision number.

5.1.2 LABEL ASSIGNMENT STATEMENT

A label assignment statement gives a variable the numerical label of another statement in the same program. Any subsequent statement with that variable automatically references the statement whose label is assigned. With READ and WRITE statements, this feature permits selection of several possible formats based on program execution.

The format is:

ASSIGN k TO i

- Where: **k** is the statement label referencing a statement in the same program unit as the assign statement
- i** is an integer variable called the assign variable

Example:

```
25 ASSIGN 20 TO IOTA
.
.
.
50 ASSIGN 30 TO IOTA
.
.
.
WRITE (3, IOTA) LIST
20 FORMAT (...)
30 FORMAT (...)
```

In the preceding example, if the program sequence includes statement 25 but skips 50, the WRITE is executed according to the FORMAT labeled 20; if the program sequence skips the statement labeled 25 but includes 50, the WRITE is executed according to the FORMAT labeled 30.

An assign variable is also used in conjunction with an assigned GO TO statement. After execution of an assignment statement, subsequent execution of an assigned GO TO statement transfers control to the statement identified by the assigned label, provided there was no intervening redefinition of that label. Used in this manner, the label must identify an executable statement.

An assign variable may be in common (Section 6.1.2) or it may be an actual argument in a procedure reference (Chapter 7). In these cases, it continues to function as an assign variable in the related program units. Thus, FORMAT statements and labels may be passed between program units.

Once it is defined in an ASSIGN statement, an integer variable may not be referenced in any statement other than an assigned GO TO statement or a formatted READ or WRITE statement until it is redefined.

5.2 CONTROL STATEMENTS

Program execution normally proceeds from statement to statement as they appear in a program. Control statements can be used to alter this sequence or cause a number of iterations of a program section. Control may only be transferred to an executable statement. A transfer to a nonexecutable statement results in a program error, which is usually recognized during compilation. With the DO statement, a predetermined sequence of instructions can be repeated any number of times by incrementing a simple integer variable after each iteration.

Statements to which control is transferred must have statement labels and they must reference executable statements within the same program as the control statement. This restriction does not apply to the assigned GO TO. The types of control statements are

GO TO	RETURN
Arithmetic IF	CONTINUE
Logical IF	Program Control
CALL	DO

5.2.1 GO TO STATEMENTS

GO TO statements transfer control unconditionally to a statement with a label whose reference is fixed or to a statement whose label is selected during execution of the program. GO TO statements may be

Unconditional GO TO

Assigned GO TO

Computed GO TO

UNCONDITIONAL GO TO STATEMENT

Execution of this statement causes the statement identified by the label to be executed next.

The format is:

GO TO k

Where: k is the statement label

Example:

```
GO TO 10
5 DIF = DIF - SUM
10 SUM = SUM + 1
```

In this program sequence, the GO TO statement causes control to skip statement 5 and execute statement 10 and those following in sequence until control is altered again. Statement 5 is not executed unless it is referenced by some other control statement in the program.

ASSIGNED GO TO STATEMENT

This statement acts as a many-branch GO TO.

The formats are:

```
GO TO i
GO TO i, (k1, ..., kn)
```

Where: i is an integer variable reference called an assign variable

k_i are optional statement labels which may be included for the programmer's convenience; they are not used by the compiler.

Before an assigned GO TO statement is executed, the current value of i must have been assigned by an ASSIGN statement. Control transfers to the statement identified by that statement label to be executed next. The i must be assigned in either the program unit of the GO TO or in another program unit where i was passed as an actual parameter or was in common.

The same integer variable reference used in the ASSIGN statement may be used in a subsequent arithmetic expression if it is re-equated to a value prior to its use in that expression.

Examples:

Format 1

```
      ASSIGN 15 TO K
      GO TO 60
15   K = 9
      L = (I**2) + K
100  ASSIGN 20 TO K
      GO TO 60
      .
      .
      .
60   CONTINUE
      .
      .
      .
      GO TO K
20   CONTINUE
```

When the program executes the ASSIGN statement, K has the statement label value 15.

Control moves to the next statement which causes a jump to statement 60, CONTINUE.

The program executes the statements following 60 in sequence until it reaches GO TO K. Since K previously has been assigned the value 15, control jumps to statement 15.

Statement 15 equates K to the value 9.

The following statement uses this value (9) of K in an arithmetic expression. (The variable reference is re-equated to a value and then used in an arithmetic expression.)

In the next statement, the program assigns 20 to K.

The next step causes a jump to 60, CONTINUE.

The program goes through the steps following 60 until it reaches GO TO K. As K has been assigned the statement label value 20, control jumps to statement 20, CONTINUE, and proceeds in sequence.

Format 2

```
ASSIGN 10 TO JUMP
.
.
.
GO TO JUMP, (5, 10, 20)
.
.
.
10 RESULT = RATE * AMOUNT
```

The program first assigns the value 10 to JUMP.

It proceeds in sequence until it encounters the GO TO JUMP statement. Since JUMP was assigned the value 10, control transfers to statement 10.

The list of labels (5, 10, 20) serves only as a check on JUMP, the variable reference. The second form operates in the same manner as the first; the list is optional.

COMPUTED GO TO STATEMENT

This form of the GO TO statement is an n-branch control transfer in which a sequence of statement labels is followed by an integer variable whose value at execution serves as an ordinal designation of the label which defines the transfer.

The format is:

```
GO TO (k1, k2, k3, ..., kn), i
```

Where: k is the statement label

i is an integer variable reference; for proper operation, i must not be specified by an ASSIGN statement

The statement identified by statement label k_i is executed next. Assume j is the value of i at the time of execution. If j ≤ 1, statement label k₁ is executed next. If j ≥ n, statement label k_n is executed next.

Example:

```
N=3
.
.
.
GO TO (100, 101, 102, 103), N
```

N is 3 and the statement number 102 is the selected control transfer.

If N were less than 1, control would be transferred to 100. If N were greater than 4, control would be transferred to 103.

5.2.2 ARITHMETIC IF STATEMENT

An arithmetic IF statement is a three-branch transfer on an arithmetic expression.

The format is

IF (e)k₁, k₂, k₃

Where: e is an arithmetic expression

k is an executable statement label. If the evaluation of e is

- Control is transferred to k₁
- ±0 Control is transferred to k₂
- + Control is transferred to k₃

Example:

IF (IOTA-6) 3, 6, 9

If the evaluation of the expression IOTA-6 produces a negative result, control transfers to the statement labeled 3; if zero, to 6; if positive, to 9.

5.2.3 LOGICAL IF STATEMENT

A logical IF statement is a two-branch transfer on a logical or relational expression.

The format is:

IF (e) s

Where: e is the logical or relational expression; upon execution of this statement, if

true Statement s is executed

false The sequence of statements following the logic IF is continued.

s is any executable statement except a DO statement or another logical IF statement

If the evaluation of e results in an overflow (integer value larger than 7FFF₁₆), unpredictable results occur. A minus zero is evaluated as equal to a plus zero.

Examples:

IF (A.EQ. 10..AND. B .EQ. 5.) GO TO 3

IF (X.GT.Z) Y = SIN (X)/2

IF (I.EQ.J) IF (L + 2) 100, 200, 300

5.2.4 CALL AND RETURN STATEMENTS

The CALL and RETURN statements establish communication between a main program and subroutines. These statements are explained in Section 7.4.

5.2.5 CONTINUE STATEMENT

The CONTINUE statement is most frequently used as the last statement in a DO loop (Section 5.2.7) to avoid terminating on GO TO or IF statements, which are illegal termination statements in DO loops.

The format is:

CONTINUE

When CONTINUE is the terminating statement of a DO loop, it causes repetition of the loop. In any other position, it serves as a do-nothing statement passing control to the next statement.

5.2.6 PROGRAM CONTROL STATEMENTS

Program control statements are STOP, PAUSE, and END.

STOP STATEMENT

This statement terminates execution of the program. Normally it is used at the end of a program. It may be used to terminate execution when an abnormal condition occurs.

When this statement is executed, the word STOP and any octal digits following it appear on the output comment device and standard output device in five-digit form.

The formats are:

STOP
STOP n

Where: n is one to five octal digits

PAUSE STATEMENT

This statement temporarily halts the execution of a program to permit checking of intermediate results. The operator enters a carriage return to resume execution with the statement immediately following PAUSE.

When this statement is executed, the word PAUSE and any octal digits following it appear on the output comment device, in five-digit form.

The formats are:

PAUSE
PAUSE n

Where: n is one to five octal digits

END STATEMENT

This statement marks the physical end of a program or subprogram. It is executable in the sense that it affects return from a subprogram in the absence of a RETURN statement, but it may not have a label.

The format is:

END

5.27 DO STATEMENT

A DO statement makes it possible to repeat a group of statements a designated number of times using an integer variable whose value is progressively altered with each repetition. The range of repetition is called the DO loop. Minimally, the DO loop consists of the DO statement with its parameters and a final statement whose label is referenced by the DO statement.

The formats are:

DO n i = m₁, m₂
DO n i = m₁, m₂, m₃

Where: n is the label of the terminal statement of the loop.

i is a positive integer variable called the control variable. With each repetition, its value is altered progressively by the increment parameter m₃. Upon exiting from the range of a DO, the control variable remains defined as the last value acquired in execution of the DO. It does not matter if the exit results from satisfying the DO or by execution of a GO TO or IF.

- m_1 is the initial parameter, the value of i at the beginning of the first loop.
- m_2 is the terminal parameter; when the value of i surpasses the value of m_2 , DO execution is terminated and control goes to the statement immediately following the terminal statement.
- m_3 is the increment parameter; the amount i is increased with each repetition.

The parameters m_1 and m_2 must be unsigned integer constants, or unsigned non-subscripted integer variables. If m_3 has the value 1, it may be omitted (first form above).

5.27.1 DO LOOP STRUCTURE

The general form of a DO loop is:

```

DO n i = m1, m2, m3
Statement 1
Statement 2
Statement 3
  ⋮
n Terminal statement

```

The range of the loop extends from the DO statement through the terminal statement, inclusively.

Statement 1, the first statement in the range of the DO, must be an executable statement.

Statements 1, 2, 3... may contain inner DO loops. This is called nesting and is explained below.

Table 5-1 shows how relationships among the DO statement parameters affect execution of a DO. The label n references the terminal statement of the DO loop, which must be an executable statement in the same program unit as the DO statement and must follow it.

The terminal statement may not be any of the following:

- GO TO
- Arithmetic IF
- RETURN
- STOP
- PAUSE
- DO
- Logical IF (if it contains any of the preceding forms)
- ASSEM (if terminal statement label is imbedded within)

Table 5-1. DO Statement Parameters

M1	M2	M3	EXAMPLE	ACTION
Integer constant	Integer constant	Integer constant	I=1, 9, 2 I=9, 1, 2 I=9, 1, -1 I=5, 6, -1 I=5, 5	Control variable is initialized. DO loop is executed at least once. Control variable is incremented.
Integer constant	Integer constant	Integer variable	I=1, 9, N I=9, 1, N I=1, 9, -N I=9, 1, -N	Completion test is made to see if loop is to be executed again.
Integer constant	Integer variable	Integer constant	I=2, J, 2 I=5, J, -2 I=5, J	Control variable is initialized.
Integer constant	Integer variable	Integer variable	I=4, J, K I=10, J, -K	Completion test is made to see if loop is to be executed.
Integer variable	Integer variable	Integer constant	I=J, K, 2 I=K, J, -2 I=K, J	Loop is executed.
Integer variable	Integer variable	Integer variable	I=M1, M2, N I=M1, M2, -N	Control variable is incremented.
Integer variable	Integer constant	Integer variable	I=J, 10, K I=J, 6, -K	
Integer variable	Integer constant	Integer constant	I=J, 5, 2 I=J, 3, -2 I=J, 10	

Example:

The following program calculates the sum of all odd numbers and the sum of all even numbers in the range of 1 to 100.

```

      IODD = 0
      IEVEN = 0
      DO 25 I = 1, 99, 2
      IODD = IODD + I
      J = I + 1
      IEVEN = IEVEN + J
25    CONTINUE
    
```

The first two steps zero out the counters for the odd and the even numbers. The DO statement initiates a loop that begins at the index value of 1 and increments in steps of 2. This series provides the odd numbers. The $J = I + 1$ statement provides the series of even numbers by adding a 1 to each of these values. The operation of this DO loop is tabulated in the following chart.

Loop	I	I _{ODD} =I _{ODD} +I	(store)	J=I+1	(store)	I _{EVEN} =I _{EVEN} +J	(store)
1	1	1=0+1	(1)	2=1+1	(2)	2=0+2	(2)
2	3	4=1+3	(4)	4=3+1	(4)	6=2+4	(6)
3	5	9=4+5	(9)	6=5+1	(6)	12=6+6	(12)
4	7	16=9+7	(16)	8=7+1	(8)	20=12+8	(20)
.
.
.

↑

Successive values
of control variable
I which is the
sequence of odd
numbers

↑

Progressive
addition of
odd numbers

↑

Sequence
of even
numbers

↑

Progressive
addition of
even
numbers

5.27.2 DECREMENTED DO LOOP

When decrementation is desired in a DO loop, the following form applies:

`DO n1 = m1, m2, -m3`

Where the value of the incremental parameter m_3 is established in a preceding statement and $m_1 > m_2$.

Example:

To find the value of N factorial (N!):

```

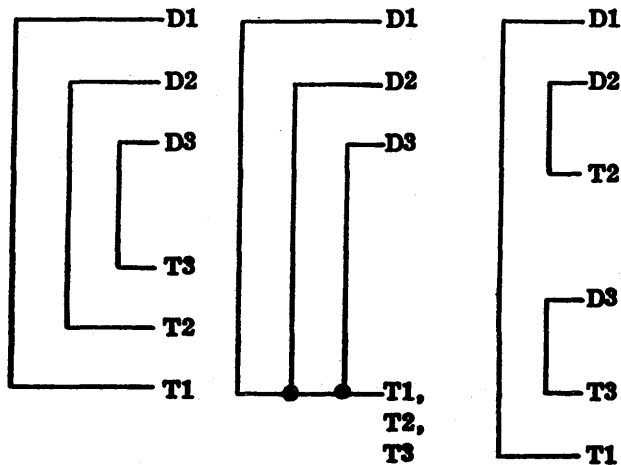
READ(1,5) N
5  FORMAT (I2)
   FACT = 1.0
   K = 1
   DO 10 I = N, 2, -K
10  FACT = FACT*I
   WRITE (3,15) FACT
15  FORMAT (F10.0)

```

5.2.7.3 NESTED DO LOOPS

DO loops may be included within DO loops as long as no inner range overlaps with any outer range. However, two or more DO loops may share the same terminal statement. DO loops may be nested up to 10 deep.

If D1, D2, and D3 are DO statements and T1, T2, and T3 are the associated terminal statements, then the following nested structures of DO loops are permitted:



Example:

This example may be used to test Fermat's Last Theorem with combinations of integer values up to 1000. The theorem states that the equation

$$X^n + Y^n = Z^n$$

is not valid for positive integer values of X, Y, and Z when n is an integer greater than 2.

Letting I, J, K equal X, Y, Z to imply integer values, the test may be programmed as follows:

```

PROGRAM FERMAT
DO 13 N = 3, 1000
DO 13 I = 1, 1000
DO 13 J = 1, 1000
DO 13 K = 1, 1000
IF (I**N+J**N-K**N) 13, 7, 13
7 WRITE (3,100) I, J, K, N
100 FORMAT (6HEUREKA/4I5)
13 CONTINUE

```

Example:

If a loan is repaid in N monthly payments with each payment equal to P and with an interest rate of R, the total repaid, S, is given by:

$$S = \frac{P}{R} \left(1 - \frac{1}{(1+R)^N} \right)$$

The following program calculates the sums repaid for monthly payments of 24, 30, and 36 months in amounts of \$20, \$30, \$40, and \$50 at interest rates 6%, 7%, 8%, 9%, and 10%.

```

DIMENSION SUM (5)
DO 30 N = 24, 36, 6
DO 20 J = 20, 50, 10
DO 10 I = 6, 10
R = I*0.01
10 SUM (I-5) = J/R*(1. - 1./((1.+R)**N))
20 WRITE (3,40) (SUM(K), K=1, 5)
40 FORMAT (5F10.2)
30 CONTINUE

```

This would print out the sums, five to a line, according to the five interest rates.

The following tabulation shows how the cycling proceeds through the DO loops, with the innermost loop varying the most rapidly and the outermost loop varying the least rapidly.

<u>Months</u>	<u>Amount</u>	<u>Rate</u>	<u>Months</u>	<u>Amount</u>	<u>Rate</u>
24	20	.06	30	40	.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
		.06			.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
30	40	.06	36	20	.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
		.06			.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
30	20	.06	30	30	.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
		.06			.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
30	30	.06	30	50	.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
		.06			.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10

5.2.7.4 DO LOOP TRANSFER

Control can be transferred within a DO loop by means of an IF or a GO TO statement, provided neither is used as a terminal statement.

The label of a terminal statement in a series of more than one DO statement may not be used in any GO TO or IF statement that occurs anywhere but in the range of the most deeply contained DO that has that terminal statement.

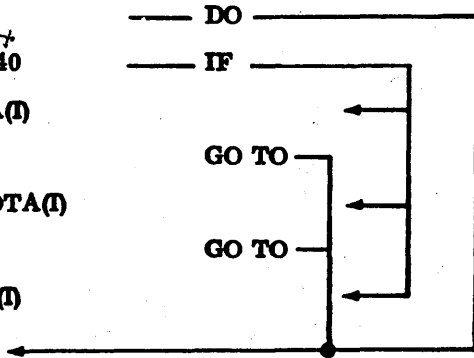
Example:

This example may be used to test 100 values for sign and accumulate three sums: negative, zero, and positive values.

```

PROGRAM TEST
DIMENSION IOTA (100)
READ (1, 10) (IOTA(I), I=1, 100)
10  FORMAT(10I5)
    INEG = 0
    IZERO = 0
    IPOS = 0
    DO 50 I = 1, 100
    IF (IOTA(I)) 20, 30, 40
20  INEG = INEG + IOTA(I)
    GO TO 50
30  IZERO = IZERO + IOTA(I)
    GO TO 50
40  IPOS = IPOS + IOTA(I)
50  CONTINUE

```



5275 EXTENDED RANGE OF A DO

If control can be transferred out of a DO loop and returned, the DO is said to have an extended range. More specifically, a DO has an extended range if it contains a GO TO or IF that can pass control outside the range of the DO and there is a GO TO or IF outside the range of the DO that can return control into the range of the DO.

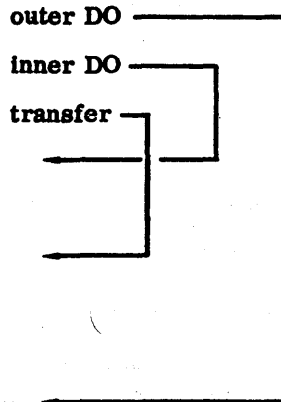
Control can be transferred from an inner DO loop to the outer DO loop that contains it. Control cannot initially pass from an outer DO loop into an inner DO loop.

Example:

This example may be used to compare two arrays of numbers and print out all sets of equivalent values.

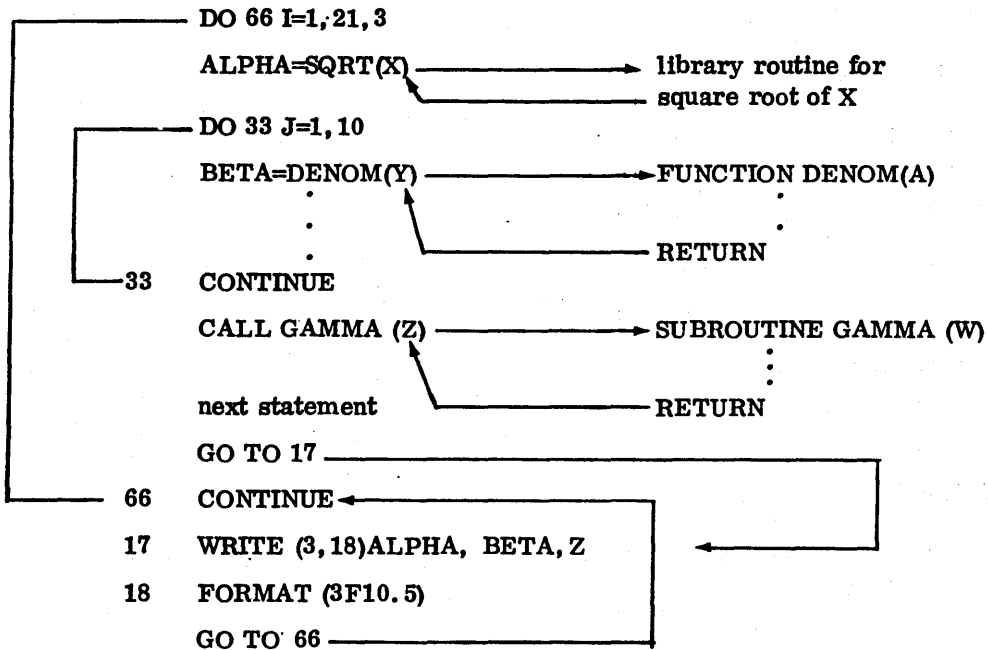
```

DO 50 I = 1,20
DO 30 J = 1,20
IF(A(I).EQ.B(J))GO TO 40
30 CONTINUE
GO TO 50
40 WRITE(3,10)A(I),B(J)
GO TO 30
10 FORMAT(F8.5,3H = ,F8.5)
50 CONTINUE
    
```



Control can be transferred out of a DO loop or nest of DO loops and returned, provided the indexing parameters are not altered and control is transferred back to the range of the same DO loop from which the exit was made.

Example:



5.3 I/O STATEMENTS

I/O statements are classified as data transfer statements and auxiliary I/O statements. The first type is the READ and WRITE statements which read records from an external unit into core and write records out of core onto an external unit. Under the second type, BACKSPACE, REWIND, and ENDFILE affect the position and check the status of external magnetic tape files; basic functions (Section 7.3.2) check the status of I/O devices. The following section applies only to the FORTRAN I/O run-time package. Consult Chapter 9 for the I/O statements that are used with other run-time packages.

The logical units defined for the various I/O operations are those defined for the MSOS system in which Mass Storage FORTRAN operates. Logical units for specific devices are likely to vary from system to system. Standard FORTRAN units should be used as much as possible:

<u>Logical Unit Number</u>	<u>Description</u>
1 CR/TAPE	Standard Input Device
2 CR/P	Standard Binary Output Device
3 PENTOL	Standard Print Output Device
4 TTY	Standard Output Comment Device

5.3.1 I/O DEVICES

The Mass Storage FORTRAN product supports all I/O devices present in MSOS. The MSOS reference manual should be consulted for specific devices.

5.3.2 MASS STORAGE FILES

There are two distinct methods of using files in MSOS. The use of file manager files is discussed in the MSOS reference manual. The material presented here applies only to FORTRAN files and must not be confused with files created via the File Manager.

Mass storage files are assigned to the scratch area of the mass storage device and are not retained after execution of a job. (Permanent files in the program library may not be defined or manipulated by FORTRAN I/O statements.) Files to be read in to or written out of mass storage must be preceded by an OPEN statement that defines the file.

5.3.3 OPEN STATEMENT

This statement provides for parameters to describe each mass storage file to be used by the program.

The formats are:

```
OPEN k,i,j,u,x  
OPEN k,i,j,u
```

Where: k is the integer name of the mass storage file to be defined; integer constant or variable.

i is the number of sectors per record; minimum record length is one sector; integer constant or variable.

j is the maximum number of records in the file; integer constant or variable.

u is the logical unit number to which file k is assigned; integer constant or variable.

x is the starting sector address[†] for file k on logical unit u; positive integer constant or variable. The sector address is assigned relative to the start of mass memory scratch.

If any of the above variables are omitted, the starting sector address is to be assigned at execution time. Subsequent mass storage files are assigned sequentially.

NOTE

If x is omitted for one mass storage file in a program, it must be omitted for all mass storage files in that program.

If x is specified for one mass storage file in a program, it must be specified for all mass storage files in that program.

Attempting to read or write a file on mass storage without defining the file by an OPEN statement results in an execution-time diagnostic and program termination. The syntax legality of the values specified by OPEN is checked at compile time.

Examples:

OPEN 35, 2, 50, 5, 1

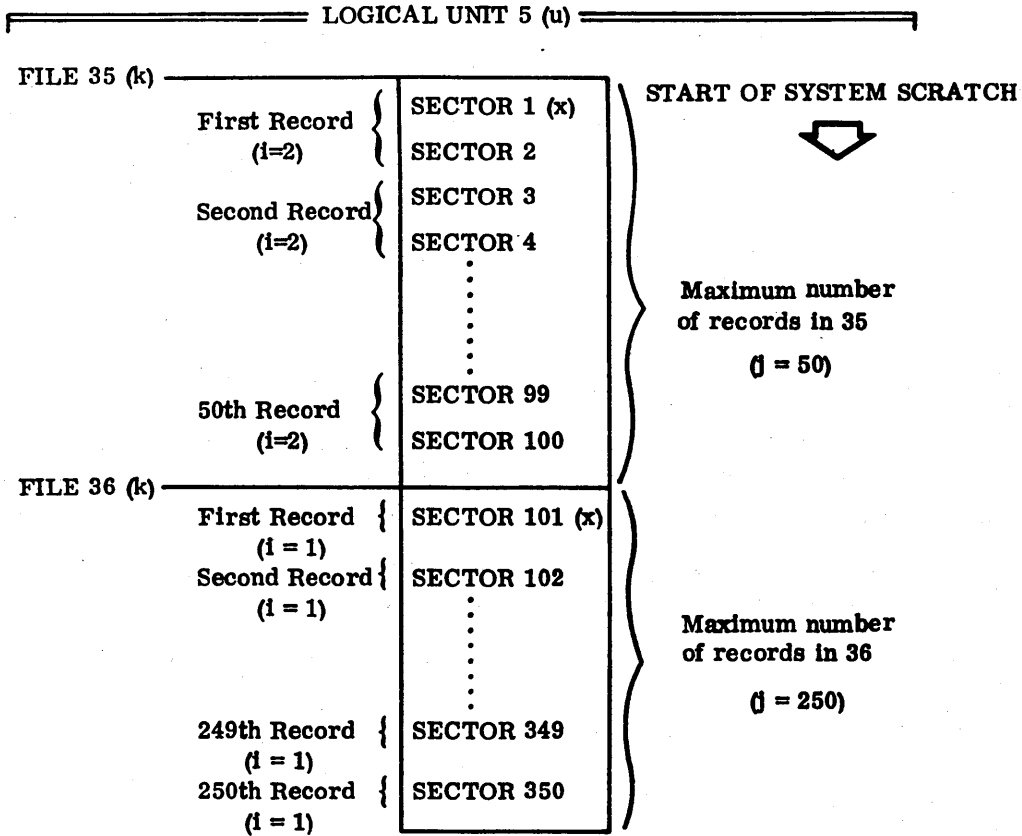
This statement defines a file referenced as 35. This file uses two sectors per record and consists of 50 records maximum (100 sectors). The file is assigned to logical unit 5 and starts at the location of the first sector relative to the start of the mass memory scratch area.

OPEN 36, 1, 250, 5, 101

This statement defines a file referenced as 36. This file uses one sector per record and consists of 250 records maximum (250 sectors). The file is assigned to logical unit 5 and starts at the location of sector 101 (relative to the start of the mass memory scratch area), which means that file 36 immediately follows file 35.

[†]The maximum number of data words in a sector is 95, since one word of each sector is for addressing that sector.

The following diagram shows the arrangement and significance of the parameter values for examples.



5.3.4 READ AND WRITE STATEMENTS

The READ and WRITE statements transfer data lists between core and external devices. These lists may include the names of variables, arrays, and array elements. The named elements are assigned values on input, and their values are transferred on output. Arrays in a list may be transferred with an implied DO of the forms:

$$i = m_1, m_2, m_3$$

$$i = m_1, m_2$$

The parameters i, m_1, m_2, m_3 are defined exactly as they are for the DO statement (Section 5.2). An implied DO does not reference a terminal statement; the range is the array to which it is applied.

Example:

```
      READ (7, 10) (A (I), I = 1,4)
10    FORMAT (F 10.6)
```

has the same effect as:

```
      DO 20 I=1, 4
20    READ (7,30) A (I)
30    FORMAT (F 10.6)
```

Both of these examples read the first value of four records into array elements A(1), A(2), A(3), and A (4). If integer variables in an input list appear as subscripts elsewhere in the list, they must appear as input variables before they appear as subscripts unless they have been previously defined.

Example:

```
      READ (7) I,J, ALPHA (I,J)
```

Data records are written in binary or ASCII. Because binary records are coded within the computer, format cannot be selected. Such records are referred to as unformatted, and they are used with magnetic tape and mass storage devices. ASCII records are used for man/machine communication and format must be specified. Such records are called formatted and can be transferred by READ and WRITE statements only when controlled by FORMAT statements (Section 6.3). In the explanations that follow, two forms of each READ/WRITE statement are given; the first applies to non-mass storage files; the second applies to mass storage files. For the ANSI option, unformatted I/O will transmit two words per integer list element. If the integer list element was typed SINGLE, then only one word is transmitted per integer list element.

Formatted READ

The format of this statement has two variations.

The first causes the input of the next record from the unit identified by lu (logical unit).

The format is

```
      READ (lu,f) list
```

Where: lu is the integer constant or non-subscripted variable reference used to identify the logical unit. MS FORTRAN assigns logical unit numbers to reference standard MSOS logical units as follows:

- 1 = Standard input (contained in \$F9)
- 2 = Standard binary output (contained in \$FA)
- 3 = Standard list output (contained in \$FB)
- 4 = Standard comment (contained in \$FC)

Logical units 5 through 99 reference actual assigned MSOS logical units of like number.

NOTE

MSOS logical units 1 through 4 cannot be referenced by FORTRAN programs unless they are also MSOS standard logical units. Actual logical unit assignments vary from system to system.

f is the format specification (Section 6.3).

label is the statement label of a FORMAT statement. The identified statement must appear in the same program unit as the I/O statement.

array is the array name which must conform to the specifications in Section 6.3.

assign is an assigned variable; the statement label assigned must be a format specification. The variable assigned may be a dummy argument or, if it is in common, it may come from another program unit.

list is a series of variables separated by commas.

The information is scanned and converted according to the format specification identified by **f**.

The resulting values are assigned to the elements specified by the list. If the list is not present, READ (lu,f), spaces over one record.

Example:

```
    READ (5,20) A,B,C
20  FORMAT (3F10.6)
```

These statements read in three floating-point values from logical unit 5 according to the FORMAT labeled 20. This specifies field widths of 10 positions with 6 decimal places.

The second format causes input of the nth record from mass storage file k.

The format is

READ (k(n),f) list

Where: **k** is the mass storage file

n is the nth record from mass storage file k

f is the format specification (see Section 6.3).

label is the statement label of a FORMAT statement. The identified statement must appear in the same program unit as the I/O statement.

array is an array name which must conform to the specifications in Section 6.3.

assign is an assigned variable; the statement label assigned must be a format specification. The variable assigned may be a dummy argument or, if it is in common, it may come from another program unit.

list is a series of variables separated by commas.

The information is scanned and converted as specified by the format specification identified by **f**. The file must have been opened by a previous **OPEN** statement.

If the file is not on a mass storage device, the request is ignored. If **f** specifies **H** (Hollerith) conversion, the record is read into the storage locations of the **FORMAT** statement replacing the **H** part of **f**.

Example:

```
OPEN 40,1,200,5
      .
      .
      .
READ (40(12),10) X
10  FORMAT (50A1)
```

These statements read the twelfth record of mass storage file 40 from logical unit 5 into array **X** which has been previously dimensioned. The record contains 50 ASCII characters according to the **FORMAT** labeled 10.

Formatted WRITE

The format of this statement has two variations.

The first writes the next record on the unit identified by **lu** (logical unit).

The format is

```
WRITE (lu,f) list
```

Where: **lu** is the integer constant or non-subscripted variable reference used to identify the logical unit. MS FORTRAN assigns logical unit numbers to reference standard MSOS logical units as follows:

- 1 = Standard input (contained in \$F9)
- 2 = Standard binary output (contained in \$FA)
- 3 = Standard list output (contained in \$FB)
- 4 = Standard comment (contained in \$FC)

Logical units 5 through 99 reference actual assigned MSOS logical units of like number.

NOTE

MSOS logical units 1 through 4 cannot be referenced by FORTRAN programs unless they are also MSOS standard logical units. Actual logical unit assignments vary from system to system.

- f** is the format specification (See Section 6.3).
- label** is the statement label of a FORMAT statement. The identified statement must appear in the same program unit as the I/O statement.
- array** is an array name which must conform to the specifications in Section 6.3.
- assign** is an assigned variable; the statement label assigned must be a format specification. The variable assigned may be a dummy argument or, if it is in common, it may come from another program unit.
- list** is a series of variables separated by commas. The list specifies a sequence of values which are converted and positioned according to the format specified by **f**. If the list is omitted, any /, nX, nH, or asterisk or quote-enclosed character strings are output until the first F, E, D, I, \$, Z, A, or R conversion is reached. If **f** does not contain /, X, or H editing characters and the list is omitted, a line of blanks is assumed.

Example:

```
WRITE (9,20) A,B,C
20  FORMAT (3F10.6)
```

These statements write the floating-point numbers from locations A, B, and C on logical unit 9.

The second format writes record n on mass storage file k.

The format is

```
WRITE (k(n),f) list
```

- Where: **k** is the mass storage file
- n** is the nth record from mass storage file k
- f** is the format specification (see Section 6.3).
- label** is the statement label of a FORMAT statement. The identified statement must appear in the same program unit as the I/O statement.
- array** is an array name which must conform to the specifications in Section 6.3.
- assign** is an assigned variable; the statement label assigned must be a format specification. The variable assigned may be a dummy argument or, if it is in common, it may come from another program unit.

list is a series of variables separated by commas. The list specifies a sequence of values which are converted and positioned according to the format specified by f. If the list is omitted, any /, nX, nH, or asterisk or quote-enclosed character strings are output until the first F, E, D, I, \$, Z, A, or R conversion is reached. If f does not contain /, X, or H editing characters and the list is omitted, a line of blanks is assumed.

If k(n) is not large enough to accommodate the converted data, truncation occurs.

Example:

```
WRITE (55(2), 10)A
10 FORMAT (F10, 2)
```

These statements write the floating-point number from location A into the second record of mass storage file 55. The file must have been opened by a previous OPEN statement.

Unformatted READ

The format of this statement has two variations.

The first form of the statement causes the input of the next record from the unit identified by lu.

The format is

```
READ (lu) list
```

Where: lu is the integer constant or non-subscripted variable reference used to identify the logical unit. MS FORTRAN assigns logical unit numbers to reference standard MSOS logical units as follows:

- 1 = Standard input (contained in \$F9)
- 2 = Standard binary output (contained in \$FA)
- 3 = Standard list output (contained in \$FB)
- 4 = Standard comment (contained in \$FC)

Logical units 5 through 99 reference actual assigned MSOS logical units of like number.

NOTE

MSOS logical units 1 through 4 cannot be referenced by FORTRAN programs unless they are also a MSOS standard logical unit. Actual logical unit assignments vary from system to system.

list is a series of variables separated by commas. If list is specified, these values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record. If the file is on a mass storage device, the request is ignored.

Examples:

READ (1) (A(I), I=1,100)

This statement reads a record from unit 1 into the storage areas specified in the DO implied list.

READ (6)

This statement skips the next record on logical unit 6. Unit 6 cannot be a mass storage device.

The second form of the statement inputs the nth record from mass storage file k.

The format is

READ (k(n)) list

Where: k is the mass storage file

n is the nth record from mass storage

list is a series of variables separated by commas. If list is specified, these values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record.

Example:

READ (31 (9)) X

This statement reads the ninth record of the mass storage file 31 into storage location X.

Unformatted WRITE

The format of this statement has two variations.

The first form of the statement creates the next record on the unit identified by lu from the sequence of values specified by the list.

The format is

WRITE (lu) list

Where: **lu** is the integer constant or non-subscripted variable reference used to identify the logical unit. MS FORTRAN assigns logical unit numbers to reference standard MSOS logical units as follows:

- 1 = Standard input (contained in \$F9)
- 2 = Standard binary output (contained in \$FA)
- 3 = Standard list output (contained in \$FB)
- 4 = Standard comment (contained in \$FC)

Logical units 5 through 99 reference actual assigned MSOS logical units of like number.

NOTE

MSOS logical units 1 through 4 cannot be referenced by FORTRAN programs unless they are also MSOS standard logical units. Actual logical unit assignments vary from system to system.

list is a series of variables separated by commas. The list may not be empty.

Example:

DIMENSION A (80), B (4)

WRITE (2) A,B

These statements write one record of two blocks on logical unit 2.

The second form writes record n on mass storage file k.

The format is

WRITE (k(n)) list

Where: **k** is the mass storage file

n is the nth record from mass storage file k

list is a series of variables separated by commas. The list may not be empty. If the number of words in the list exceeds the record size specified for k, an execution time diagnostic is given and the program terminates.

Example:

```
DIMENSION A(10)
OPEN 22, 1, 500, 8
  ⋮
WRITE (22(100)) A
```

These statements write ten words from array A into the 100th record of mass storage file 22 on logical unit 8.

5.3.5 AUXILIARY I/O STATEMENTS

Auxiliary I/O statements are applicable only to files residing on magnetic tape. The following section applies only to the FORTRAN I/O run-time package.

REWIND Statement

This statement positions the unit identified by lu at its loadpoint.

The format is

```
REWIND lu
```

Where: lu is the integer constant or non-subscripted variable reference used to identify the logical unit. MS FORTRAN assigns logical unit numbers to reference standard MSOS logical units as follows:

- 1 = Standard input (contained in \$F9)
- 2 = Standard binary output (contained in \$FA)
- 3 = Standard list output (contained in \$FB)
- 4 = Standard comment (contained in \$FC)

Logical units 5 through 99 reference actual assigned MSOS logical units of like number.

NOTE

MSOS logical units 1 through 4 cannot be referenced by FORTRAN programs unless they are also a MSOS standard logical unit. Actual logical unit assignments vary from system to system.

BACKSPACE Statement

This statement causes the unit identified by lu to go back to the beginning of the preceding block. If the unit identified by lu is positioned at its load point, an error diagnostic is printed and the program is terminated. If a record contains n blocks of data, then n BACKSPACEs must be specified to skip backwards over that record.

The format is

BACKSPACE lu

ENDFILE Statement

This statement causes an endfile record to be recorded on the unit identified by lu. The endfile record is a unique record signifying a demarcation of a sequential file. The EOF function (Table 7-4) is used to determine if an endfile record has been encountered during execution of a READ statement. After reading an ENDFILE and before reading again from that unit, a test must be made for the END FILE using the EOF function.

The format is

ENDFILE lu

Where: lu is the integer constant or non-subscripted variable reference used to identify the logical unit. MS FORTRAN assigns logical unit numbers to reference standard MSOS logical units as follows:

- 1 = Standard input (contained in \$F9)
- 2 = Standard binary output (contained in \$FA)
- 3 = Standard list output (contained in \$FB)
- 4 = Standard comment (contained in \$FC)

Logical units 5 through 99 reference actual assigned MSOS logical units of like number.

NOTE

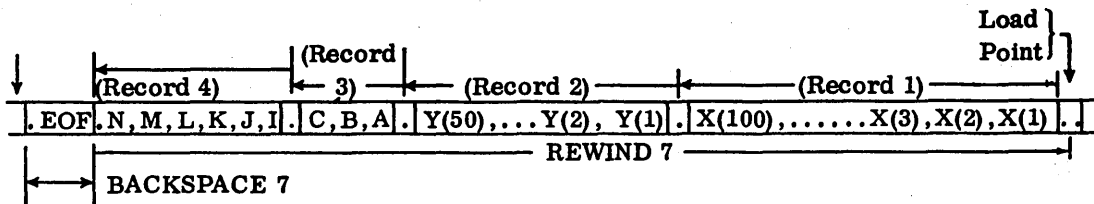
MSOS logical units 1 through 4 cannot be referenced by FORTRAN programs unless they are also a MSOS standard logical unit. Actual logical unit assignments vary from system to system.

Example:

```
WRITE(7) (X(I), I=1,100)           (record 1)
WRITE (7) (Y(I), I=1,50)           (record 2)
WRITE (7) A, B, C                   (record 3)
WRITE(7) I, J, K, L, M, N          (record 4)
ENDFILE 7
BACKSPACE 7
REWIND 7
```

This sequence of auxiliary I/O statements would write and move the tape of logical unit 7 as illustrated.

ENDFILE 7



5.3.6 TAPE RECORDS AND BLOCKS

Tape records and blocks may be binary or ASCII. Binary tape records (paper or magnetic tape) are composed of 86-word blocks. The first word of a binary block, the control word, is followed by 85 data words. Records may be one per block or extend over several blocks. A block may not contain more than one record.

The control word is zero for all blocks except the last, where it equals the number of blocks in the record.

ASCII tape records (paper or magnetic tape) are composed of blocks with a maximum of 68 words. Each block is one record. Output statements which write more than 68 words (136 characters) in a record are truncated to 68-word records. Input statements which read more than 68 words (136 characters) result in diagnostics and program termination.

Paper Tape

A binary block is preceded by a header word which contains the block size in one's complement form. It is followed by a checksum word. The checksum added to the sum of the data words and the header word equals zero. Overflow is ignored when computing the checksum.

Example:

```
WRITE (2) (A(I), I = 1,250)
```

This request produces a record on paper tape with the following format.

Header word	}	Block 1
Control word = 0		
85 words of data		
Checksum		
Header word	}	Block 2
Control word = 0		
85 words of data		
Checksum		
Header word	}	Block 3
Control word = 3		
80 words of data plus 5 empty words		
Checksum		

An unformatted (binary) read transmits one record with the format produced by an unformatted (binary) write. The header word and checksum are not transmitted to the buffer.

Magnetic Tape

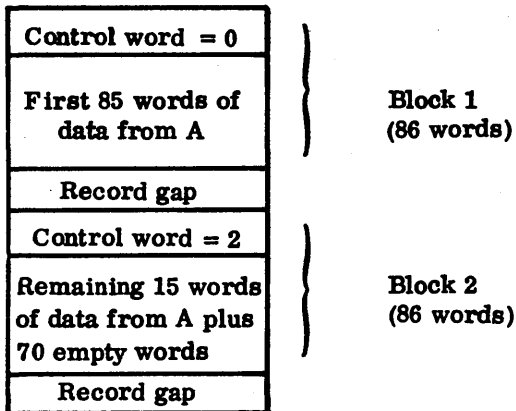
All binary blocks and ASCII blocks on magnetic tape are followed by a record gap.

ASCII is converted to extended BCD before output on magnetic tape and converted from extended BCD to ASCII when input from magnetic tape.

Example:

```
DIMENSION A(100)
WRITE (6) (A(I), I=1,100)
```

This request produces the following binary record on magnetic tape.



If no list is specified, the binary read request skips one block. Regardless of the length of the record, only the number of words specified in each list is transmitted; the remainder of the record is skipped.

5.3.7 MASS STORAGE RECORDS AND SECTORS

Mass storage records may be binary or ASCII. Binary records on disk or drum are composed of 96-word sectors. The control word (first word of a sector) is followed by 95 data words. Records may be one per sector or may extend over several sectors. A sector may not contain more than one record.

Mass storage ASCII records are subject to the same restrictions as ASCII records on tape. Each record may contain a maximum of 68 words (136 characters). Statements which output more than 68 word records cause the record to be truncated to 68 words. Statements that input records exceeding 68 words result in a diagnostic and program termination.

Example:

```

DIMENSION A(150),B(180)
OPEN 2,2,25,5,1
WRITE (2(5))A
WRITE (2(6))B
    
```


These requests produce the following binary records on disk or drum.

Control word = 0	}	Sector 8
Record 5 (95 words of binary data from array A)		
Control word = 2	}	Sector 9
(55 words of binary data from array A)		
Control word = 0	}	Sector 10
Record 6 (95 words of binary data from array B)		
Control word = 2	}	Sector 11
(85 words of binary data from array B)		

A binary read transmits the number of words specified in the list; the list must not specify more words than the record contains. If no list is specified, the binary read request is ignored.

5.3.8 PRINTING OF CARRIAGE CONTROL

The first character of a formatted record is not printed if the print unit is the FORTRAN line printer. The first character which can appear as a single Hollerith character (for example, 1H0), determines vertical spacing on I/O printer units as follows:

<u>Character</u>	<u>Vertical Spacing Before Printing</u>
0	Two lines
1	To first line of next page
+	No advance
Other	One line

Consult the MSOS Reference Manual for specific characteristics of each character output device driver.

5.4 ASSEM STATEMENT

The ASSEM statement provides communication with the operating system or the core-resident programs, or in-line coding (not possible with FORTRAN statements). With ASSEM, in-line code can be compiled in the form of absolute constants, relative address constants, and absolute address constants. Statements cannot be subscripted. Each parameter generates one word (16 bits) of code, except for statement labels and control indicators.

The format is

ASSEM p_1, p_2, \dots, p_n

Where any p_i may be

1. Hexadecimal constant of the form
 \$HH...H
 where each H is a hexadecimal digit (up to 4 digits).
2. Absolute address constant of the form
 +AA...A(c)
3. Self-relative address constant of the form
 AA...A(c)
4. $.n_1 \dots n_j$
 where n_1 through n_j is the statement label, $j \leq 5$ of the next p_i .
5. Control indicator for relative address constants which are to be other than self-relative of the form
 *
6. Relative address constant which is other than self-relative of the form
 *AA...A(c)
7. Relative indirect address constant which is other than self-relative of the form
 *(AA...A(c))

In 2, 3, 6, and 7, AA...A may be a variable, an array name, or a statement label. The c is an integer constant greater than zero designating an element within the array if AA...A is an array name. External names are permitted only in forms 2 and 3 and, if used, must be declared in an EXTERNAL statement (Section 7.4.3). Subscripts are ignored.

The ASSEM statement produces a string of successive constants in the program in the order specified. Hexadecimal constants can be used to specify data or instructions. When used to generate instructions, the operation code, indirect flags, relative flags, indices, and delta value are coded in hexadecimal. The self-relative address constant produces a value equal to the distance between the location of the variables in the program and the location to the address constant (the positive direction is from smaller to larger address).

Relative address constants, other than self-relative (forms 6 and 7), are special forms to create calling sequences to the operating system (refer to the MSOS reference manual). They are created as distance relative to the last occurrence of the control indicator (form 5). These address constants, along with associated control indicators, must appear on the same or consecutive ASSEM statements (no intervening FORTRAN statements are allowed). Forms 6 and 7 are illegal if no control indicator (form 5) has been encountered.

Example 1.

```
DIMENSION K(3)
ASSEM      $C0C5,$6400,+K(1)
```

These statements produce the equivalent of the following code at the point of the ASSEM statement:

<u>Operation Code</u>	<u>Address</u>
LDA-	\$C5
STA+	K

Example 2.

```
ASSEM      .12,$C800,K(2)
```

This statement produces the equivalent of the following code:

<u>Statement Label</u>	<u>Operation Code</u>	<u>Address</u>
12	LDA	K + 1

The address (K+1) in this example is relative to the current location counter. The statement label may be referenced from anywhere within the bounds of the program unit, except as a DO loop statement terminator.

The following statements are syntactically incorrect:

```
ASSEM      102 (2)
ASSEM      (x)
```

Example 3.

This example shows how an operating system request can be generated with the ASSEM statement.

For the macro instruction

FREAD 1,ca,sa,wc,m,cp,a,x,d

the macro assembler generates the following command sequence output:

	15	14	13	12	11	10	9	8	7		4	3	0
Word 0	RTJ - (\$F4)												
1	0	d	rc				x	rp			cp		
2	ca												
3	thread												
4	ec		m	a	l								
5	wc or wca												
6	sa												

If the following values are assigned

<u>Parameter</u>	<u>Meaning</u>	<u>Value</u>
d	Part 1 request indicator	0
rc	Request code	4
rp	Request priority	0
cp	Completion priority	1
ca	Completion address	Statement label 1000
thread		Value 0
ec	Error code upon completion of request	0
m	Mode	0
a, l	Pointer to unit	1, \$F9 (unit is system input)
wc	Word count	ICOUNT
sa	Starting address	ISTART

And if x is 1, then

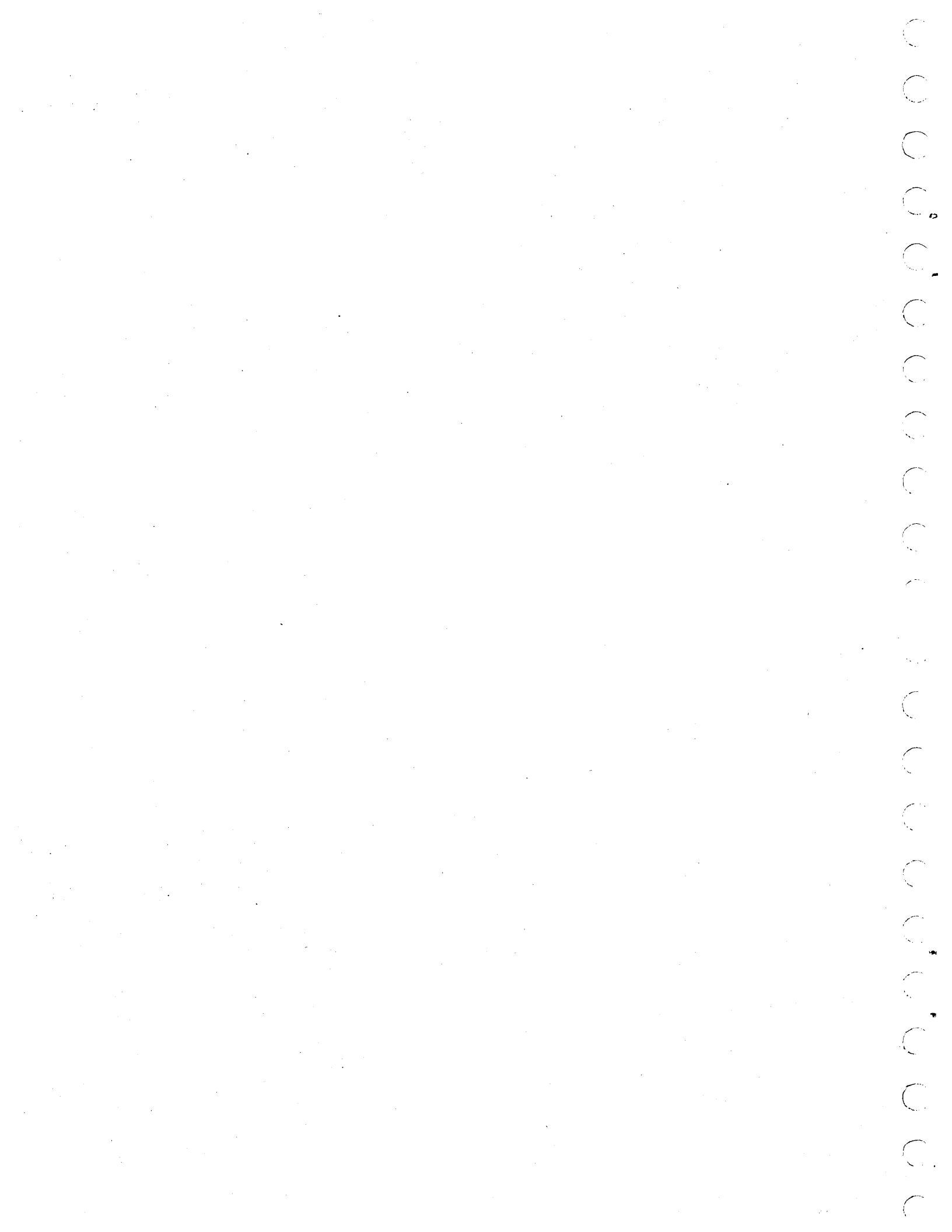
ca	Completion address	ca + address of WORD1
wca	Word count address	(wca) + address of WORD1
sa	Starting address	sa + address of WORD1

If sa is enclosed in parentheses, it produces an indirect address for the starting address.

(sa) location of starting address = (sa)+(address of WORD1).

The parameter list begins at the statement label 1001. An ASSEM statement simulates the macro instruction FREAD with the parameters previously described in either of two ways:

1. Using a relative address to reference the starting address:
ASSEM \$54F4, *, . 1001, \$0901, *1000, \$0, \$08F9, *(ICOUNT), *ISTART
2. Using a relative indirect address to reference the starting address:
ASSEM \$54F4, *, . 1001, \$0901, *1000, \$0, \$08F9, *(ICOUNT), *(ISTART)



Nonexecutable statements provide the compiler with information regarding data structure and storage; they perform no action in the execution of a program. Nonexecutable statements are as follows:

Specification

Data initialization

FORMAT

Function defining (refer to Chapter 7)

Subprogram (refer to Chapter 7)

6.1 SPECIFICATION STATEMENTS

Specification statements specify type, word structure, and storage for variables. These statements are as follows:

DIMENSION

COMMON

EQUIVALENCE

EXTERNAL (refer to Chapter 7)

RELATIVE (refer to Chapter 7)

Type

Byte

6.1.1 DIMENSION STATEMENT

Before an array can be used in a program, sequential storage locations must be reserved for all its elements, usually through the **DIMENSION** statement.

The format is:

DIMENSION $v_1(i_1), v_2(i_2), \dots, v_n(i_n)$

Where: v is an array name

i is one, two, or three subscripts giving the maximum dimensions of the array (refer to Appendix G for a detailed treatment of arrays).

Example:

For an array IOTA with three rows, four columns, and five planes, the statement would be

```
DIMENSION IOTA (3, 4, 5)
```

6.1.2 COMMON STATEMENT

Through the COMMON statement, variables in a subprogram (Section 7.4) reference the same storage locations as variables in the main program. In this way, the subprogram can make use of data blocks that are a part of the main program without the use of dummy arguments (Section 7.1).

The format is:

```
COMMON /x/a1,a2,...,an
```

Where: x is a symbolic name identifying a block of storage. This block is called a labeled common block; only one such block may appear in a program. If x is missing, the block is referred to as a blank common block and the pair of slashes may be omitted. Only one blank block and one labeled common block may appear in a program.

a is a list of simple variables and arrays (subscripted and unsubscripted). These are stored sequentially in each block in the order of their appearance.

Although a program may only have one labeled common block and one blank common block, variables and arrays can be assigned to these blocks by any number of COMMON statements, both labeled and blank. In addition, a single COMMON statement may contain labeled and blank assignments in any order. In all cases, the lists are stored in appropriate blocks in the order of their appearance. A list a_i may not contain formal arguments. If a nonsubscripted array name appears, the dimensions must be defined by a DIMENSION statement in that program unit. Arrays may be dimensioned in the COMMON statement by a subscript string following the array identifier. If an array is dimensioned in both a COMMON statement and a DIMENSION statement, a compiler diagnostic results.

Items in labeled common may be preset with initial values in the BLOCK DATA subprogram (Section 7.4.7).

Consult the MSOS Reference Manual for system organization when blank and/or labeled common is used in a system.

The length of a common block is determined by the number and type of the list identifiers. The length of common block AB in Figure 6-1 is the sum of the length of the items in the labeled blocks. LIST is

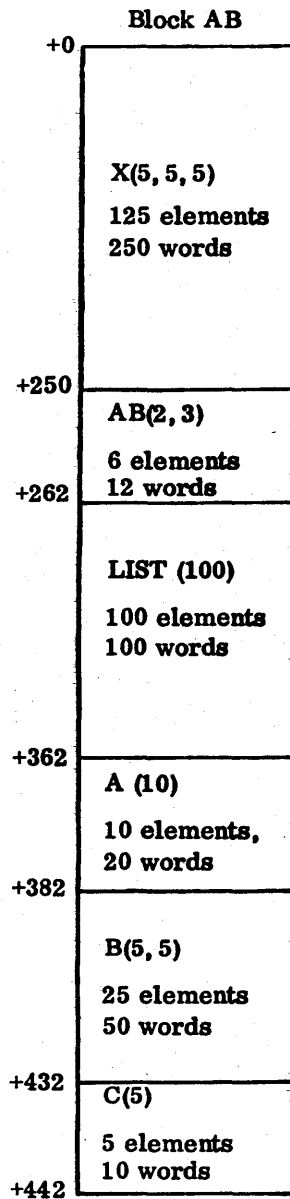


Figure 6-1. Common Block

the only integer array; and, assuming it is type SINGLE, each element requires only one word. The other arrays are real and require two computer words for each element. The total number of words in block AB is 442.

Another block of common is formed from the elements in blank or unlabeled common: A1, B1, C1, D, E, and F. The length of blank common depends on the dimensions of A1, B1, C1, and E which are not specified in the COMMON statements. If no DIMENSION statement appears for any of these identifiers, they are assumed to be simple variables.

Example:

```
COMMON/AB/X(5, 5, 5), AB(2, 3), LIST(100)
COMMON/AB/A(10)/AB/B(5, 5), C(5)
COMMON//A1, B1, C1, D(10, 10)
COMMON E, F(10, 10, 10)
```

The various program units executed together need not declare the same size common storage block, but the first program unit to be loaded must declare the largest block.

To meet ANSI standards, a real data element and an integer data element must occupy the same number of storage units to accommodate mixed-mode extensions of common through the EQUIVALENCE statement. To maintain ANSI compatibility, MS FORTRAN includes the ANSI option of allocating two words of storage for each integer data element. Only the first word is used in computation. If an integer is type SINGLE (Section 6.1.4), it always occupies one word of storage.

6.1.3 EQUIVALENCE STATEMENT

This statement makes it possible for different variables in a single program to reference the same storage location. The difference between COMMON and EQUIVALENCE is that COMMON assigns variables in different programs to the same storage locations, whereas EQUIVALENCE assigns variables in the same program to the same locations.

The format is:

```
EQUIVALENCE (a1, b1, ...), (a2, b2, ...), ..., (an, bn)
```

Where: (a₁, b₁, ...) is an equivalence group of two or more simple variables or subscripted array elements sharing a single location

A multisubscripted array element can be represented as a singly subscripted variable with the formula

$$a + A*(b-1) + A*B*(c-1)$$

which gives the ordinal location on an element with subscript (a, b, c) in an array whose maximum dimensions are (A, B, C). The formula is explained in Appendix G.

Example:

If an array element of a three-dimensional array were to be referenced with a single-dimensional array element:

```
DIMENSION I(3, 4, 5), K(60)
EQUIVALENCE (I(1, 1, 1), K(1))
```

then element I (2, 1, 4) may be referenced as K(38) by using the array successor function.

$$2 + 3*(1-1) + 3*4*(4-1) = 38$$

The manner in which storage is allocated to equivalenced arrays depends on whether the storage area is a common block or not. The EQUIVALENCE statement does not rearrange common, but arrays may be defined as equivalent so that the length of the common block is extended.

However, the origin of a common block may not be changed by an EQUIVALENCE statement. When two variables or array elements share storage because of the effects of EQUIVALENCE statements, both their symbolic names may not appear in COMMON statements in the same program, because COMMON stores elements in serial order as they appear, making it impossible for any two elements to share the same storage.

Examples:

If two arrays, not in common, are equivalenced according to the following definitions:

```
INTEGER A, B, C
DIMENSION A(3), B(2), C(4)
EQUIVALENCE (A(3), C(2))
```

Then storage locations are assigned as follows:

L	A(1)	
L+1	A(2)	C(1)
L+2	A(3)	C(2)
L+3		C(3)
L+4		C(4)
.		
.		
.		
M	B(1)	
M+1	B(2)	

However, when one of the arrays in common is used in an EQUIVALENCE statement

```
COMMON A(3), B(2)
EQUIVALENCE (B(2), C(2))
```

storage locations are assigned as follows:

L	A(1)	
L+1	A(2)	
L+2	A(3)	
L+3	B(1)	C(1)
L+4	B(2)	C(2)
L+5		C(3)
L+6		C(4)

EQUIVALENCE statements may be written

EQUIVALENCE (a1, b1)
EQUIVALENCE (a2, b2)

where the variable names or the array element names are paired. Up to 51 EQUIVALENCE statements of this form are permissible. This limits the number of equivalenced names to 102. More variable or array element names can be used if multiple names are included in one EQUIVALENCE statement. The maximum number is 100 names in a statement of the form EQUIVALENCE (a₁, a₂, a₃, ..., a₉₈, a₉₉, a₁₀₀). The number of statements must be reduced if this form is used.

An optimum declaration is six statements of the paired form with one statement of the multiple name form, resulting in the equivalencing of 112 names.

Formal parameters specified in SUBROUTINE or FUNCTION statements may be referenced in EQUIVALENCE statements (Section 7.4).

6.1.4 TYPE STATEMENT

To override or confirm implicit typing of a symbolic name, a type statement is used. It may supply dimension information for arrays.

The format is

$t v_1, v_2, \dots, v_n$

Where: t is INTEGER, REAL, DOUBLE PRECISION, or SINGLE

v_1 is a variable name, function name, array name, or an array declarator

If a symbolic name appears in a SINGLE statement the associated data is typed as integer. Dimension information may be given. When the ANSI option has been declared, appearance of a name in a SINGLE statement declares that each data element specified occupies a single storage unit.

Examples:

INTEGER ALPHA	(Over-ride)
REAL BETA	(Confirming)
DOUBLE PRECISION DELTA	(Over-ride)
SINGLE IOTA (3, 6)	(Confirms the integer type and gives dimensions of the array named)

6.1.5 BYTE STATEMENTS

Byte statements make it possible to reference a segment of an integer variable.

The format is

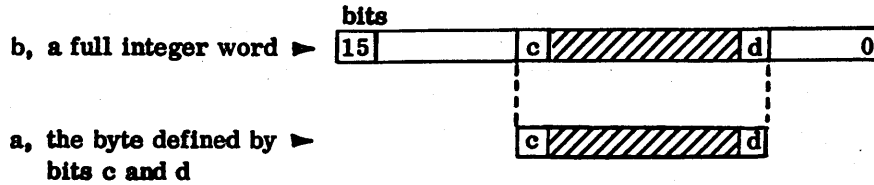
$$t(a_1, b_1(c_1=d_1)), \dots, (a_n, b_n(c_n=d_n))$$

Where: t is BYTE or SIGNED BYTE

a is the name of the byte (either an integer variable or an integer array)

b is the integer from which a is derived (integer variable, integer array, or an integer array element)

c/d are upper and lower bits of b that define a as illustrated:



c and d are positive integer constants with the range $15 \geq c \geq d \geq 0$

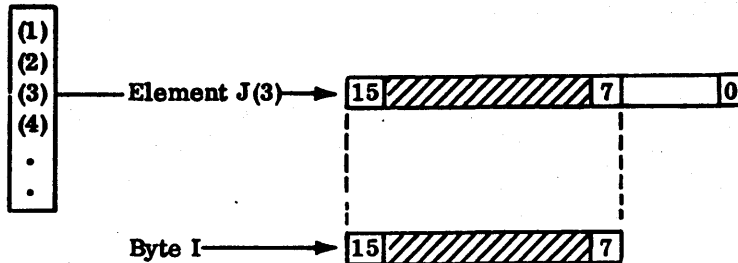
All arrays must be previously dimensioned (Appendix G).

The simplest forms of the byte statements are the cases where a is an integer variable and b is an integer variable or an integer array element.

Example:

BYTE (I, J(3) (15=7))

Array J

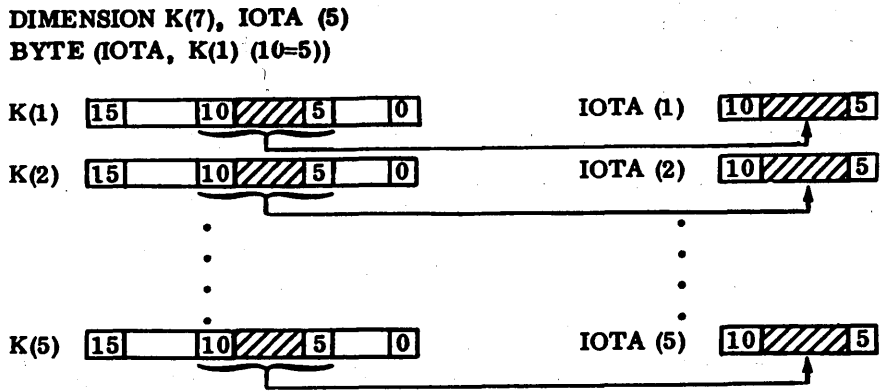


Byte I is defined as the segment from bit 15 to bit 7 of the third element of array J

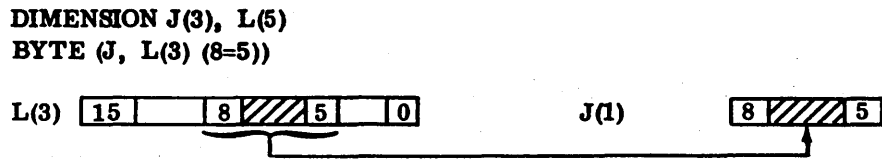
When a is an integer array, b must be an integer array or an integer array element.

Examples:

1. With b an array:



2. With b an array element:



If a_i is an array name, then each element of the array is such a byte of the corresponding element of b_i . The statement acts as an EQUIVALENCE, extending the size of b_i as much as is necessary to accommodate a_i ; the number of byte statements permissible follows the same rules as the EQUIVALENCE statements.

If t is BYTE, a_i is treated as a positive integer. The exception is if $c_i = 15$ and $d_i = 0$. In this case, a_i is treated as signed.

If t is SIGNED BYTE, a_i is treated as a signed integer. A signed byte of a single bit is treated as zero. The byte is stored in one's complement form. The high order bit is thus a sign bit. In both cases, a_i is type integer.

A byte variable or array is treated as an integer variable or array in the list of an I/O statement, as an argument for an intrinsic function, or as a parameter in the subroutine or function statement calls.

Formal parameters specified in SUBROUTINE or FUNCTION statements may be referenced in BYTE statements (Section 7.4).

6.2 DATA STATEMENT

The DATA statement is used to assign constant values to variables or arrays at the time of compilation; therefore, it is not executable.

The format is

$$\text{DATA } k_1/d_1/, k_2/d_2/, \dots, k_n/d_n/$$

Where: k is a list containing names of variables, arrays, array elements, and implied DO loops.
 d is a list of optionally signed numeric constants or literal constants, any of which may be preceded by J^* . When the form J^* appears before a constant, it indicates that the constant is to be repeated J times. J must be an integer constant. There must be a one-to-one correspondence between the list-specified items and the constants.

If an array or elements of an array are to be assigned values by a DATA statement, the array must have been previously dimensioned and each element may be listed separately in the DATA statement.

The DATA statement may be used with labeled common but not with blank common. The list k may not include byte or dummy arguments (Section 7.1). Values assigned may be redefined during execution, but not by a DATA statement, since its action terminates at compile time.

Arrays may be assigned values with an implied DO of the form:

$$\begin{aligned} &(A(I), I=i_1, i_2) && \text{or } (A(I), I=i_1, i_2, i_3) \\ &((A(I, J), I=i_1, i_2), J=j_1, j_2) && \text{or } ((A(I, J), I=i_1, i_2, i_3), J=j_1, j_2, j_3) \\ &(((A(I, J, K), I=i_1, i_2), J=j_1, j_2), K=k_1, k_2) && \text{or } (((A(I, J, K), I=i_1, i_2, i_3), J=j_1, j_2, j_3), K=k_1, k_2, k_3) \end{aligned}$$

Where: A is the name of a previously defined array.

I, J, K are the names of integer variables. The order of the subscripts must be maintained. Constants are not allowed as subscripts.

i_1, i_2, i_3 is a non-zero positive integer constant for initial value; may not be greater than limit value.

j_1, j_2, j_3 is a non-zero positive integer constant for limit value; may not be greater than the array's previously defined dimension.

k_1, k_2, k_3 is an optional non-zero positive integer constant for increment value; equals 1 if not expressed.

Implied DO loop examples:

```
DATA (A(I), I=1, 5) /1.0, 2.0, 3.0, 4.0, 5.0/
```

```
DATA ((A(I, J), I=1, 3, 2), J=1, 5, 2) /2*1.0, 2*2.0, 3.0, 4.0/
```

```
DATA (A(I), I=1, 10, 2), (B(J), J=1, 4) /9*'ABCD'/, K/8/
```

If the implicit type of a variable does not agree with its declared type, for example, (REAL I), then a DATA statement assigning a value to that variable must appear after the type is declared.

Examples:

```
Illegal    DATA I/3.5/  
           REAL I
```

```
Legal      REAL I  
           DATA I/3.5/
```

LITERALS IN DATA STATEMENT

Numeric constants are right-justified with leading zeros and literal constants are left-justified with trailing blanks.

The simple general form is:

```
DATA R, I/'AAAA', 'BB'/
```

This stores \$4141 \$4141 into R and \$4242 into I.

The alternate equivalent form is:

```
DATA R/'AAAA'/, I/'BB'/
```

There must be a matching total set of data for the total elements specified.

```
Illegal    DATA R, I/'AAAA'/'BB'/
```

```
Legal      DATA R, I/'AA', 'B'/
```

This stores \$4141 \$2020 into R and \$4220 into I.

The number of literal characters per element is as follows:

Integer 2 characters
Real 4 characters
Double Precision 6 characters

DATA I(1),I(2),I(3)/3*'AB'/
 or
DATA I/3*'AB'/
 or
DATA (I(K),K=1,3)/3*'AB'/

This stores \$4142 \$4142 \$4142 into array I.

Blank within quotes, as well as other legal characters allowed by the compiler except another quote, will be stored in their corresponding ASCII hexadecimal value.

6.3 FORMAT STATEMENT

The following section applies to the FORTRAN I/O run-time package only.

Formatted READ and WRITE statements must be accompanied by a FORMAT statement which defines the field width and data type of each element in the I/O list. The whole set enclosed in parentheses is the format specification.

The format is:

FORMAT (q₁t₁z₁t₂z₂...t_nz_nq₂)

Where: q is a set of one or more slashes or is empty
 t is a field descriptor or a group of field descriptors
 z is a field separator which is either a comma or a slash

Some representative combinations of the FORMAT statement form are the following:

(q ₁	t ₁	z ₁	t ₂	z ₂	t ₃	z ₃	t ₄	z ₄	q ₂)
(3I5	,	E10.2	,	2F10.4)
(//		5F10.5	,	3I1	/	10I10	,	E12.6)
(3A3	,	A1	,	3R3	,	R1		//)
(1H0	,	5E12.6	,	3HEND)

The **FORMAT** statement is nonexecutable. It must appear in the same program unit as the **I/O** statement and it must have a statement label. The label may be assigned in an **ASSIGN** statement. The assign variable and label can be from a different program unit. The type of field descriptor determines the type of specification. A conversion specification contains field descriptors for converting information; an editing specification contains field descriptors for editing information.

6.3.1 FIELD DESCRIPTORS

The format field descriptors are of the following forms:

rFw.d	Single-precision floating-point without exponentiation	
rEw.d	Single-precision floating-point with exponentiation	
rDw.d	Double-precision floating-point with exponentiation	
riw	Decimal integer conversion	
r\$w or rZw	Hexadecimal conversion	
rAw	Alphanumeric conversion	
rRw	Alphanumeric conversion	
nHh₁h₂...h_n	Heading and labeling	} Editing specifications
nX	Spacing factor	
Asterisk	*String of ASCII characters*	
or		
Quote	'String of ASCII characters'	

1. The symbols **F**, **E**, **D**, **I**, **\$**, **Z**, **A**, **R**, **H**, and **X** are the conversion codes; they indicate the manner of conversion and editing between the internal and external representation.
2. **w** and **n** are non-zero integer constants representing the width of the field in the external character string.
3. **d** is an integer constant representing the number of digits in the fractional part of the external character string.
4. **r**, the repeat count, is an optional non-zero integer constant indicating the number of times to repeat the succeeding basic field.
5. Each **h_i** is one of the characters in the FORTRAN character set.

For all descriptors other than Hollerith literals, the field width must be specified. For descriptors of the form **w.d**, the **d** must be specified, even if it is zero. Furthermore, **w** must be greater than or equal to **d**. The number of characters produced by an output conversion should not exceed the field width. If so, data is not transferred and the output field is filled with asterisks.

The phrase basic field descriptor will be used to signify the field descriptor unmodified by **r**.

6.3.2 FIELD SEPARATORS

The format field separators are the slash and the comma. A series of slashes is also a field separator. Field descriptors or groups of field descriptors, except the H and X descriptors, are separated by a field separator.

6.3.3 NUMERIC CONVERSION

The numeric field descriptors Iw, Fw.d, Ew.d, Dw.d, Zw and \$w are used to specify the I/O of integer, real, and double-precision data. The following rules apply to all numeric conversions.

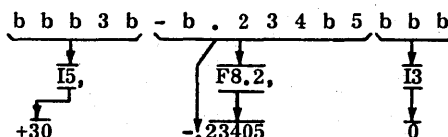
1. Leading blanks are not significant and other blanks are zeros. Plus signs may be omitted. A field of blanks is zero.
2. In input conversion of floating-point numbers, a decimal point in the input field overrides the decimal point specification in the FORMAT statement.

Examples:

Values punched
in card (b =
blank or space)

FORMAT

Read as



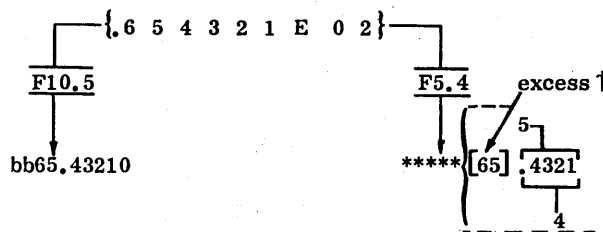
3. In numeric output conversions, the output field is right-justified and blanks are inserted if the number of characters produced by the conversion is less than the specified field width.

Examples:

Value stored

FORMAT

Printed as



4. If the conversion produces a negative value, the output field will be signed. A positive value will be unsigned, except for the exponent in an E or D conversion which will always be signed.
5. If the number of characters produced is greater than the field width, the data is not transferred and the output field is filled with asterisks.

† Only asterisks are printed because the number is too large for the field.

6.3.3.1 INTEGER CONVERSION

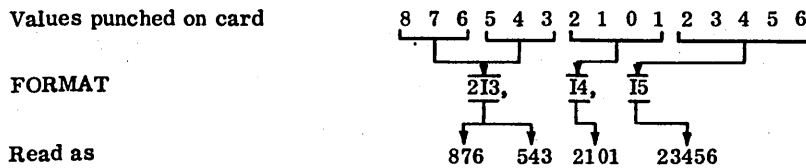
The specifications for integer conversion are Iw for decimal and \$w or Zw for hexadecimal.

Iw INPUT

Iw specification is used to input decimal integer values. The input field consists of an integer subfield and may contain only the characters +, -, 0 through 9, or blank. When a sign appears, it must precede the first digit justified in the specified variable.

Blanks are interpreted as zeros. The value is stored right-justified in the specified variable.

Example:



Iw OUTPUT

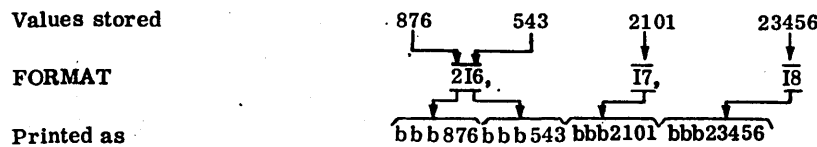
Iw specification is used to output decimal integer values; the corresponding list element must be a decimal integer quantity. The output quantity occupies w output record positions right-justified in the field w as

$\Delta s d, . . . , d$

Where: Δ is a possible blank fill
 s is the sign; minus if negative, blank or suppressed if positive.
 d, . . . , d are the most significant decimal digits of the integer (maximum absolute value is 32,767).

If the field w is larger than the number required, the output quantity is right-justified with blank fill on the left. If the field is too short, it is filled with asterisks.

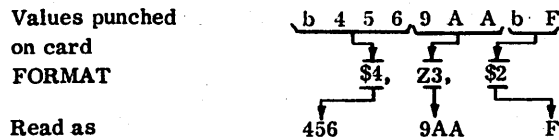
Example:



\$w OR Zw INPUT

\$w or Zw specification is used to input hexadecimal integer values. The input field w consists of a string of hexadecimal integer characters; blanks are interpreted as zero.

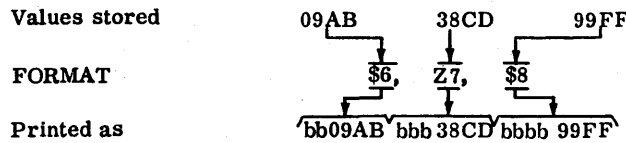
Example:



\$w OR Zw OUTPUT

\$w or Zw specification is used to output hexadecimal integer values. The output quantity occupies w output record positions right-justified in the field w. It is an unsigned hexadecimal integer value with a maximum absolute value of FFFF, containing leading zeros not exceeding the fourth significant hexadecimal position. If the field is too short to output the leftmost non-zero digit, the field is filled with asterisks. If real or double precision variables are referenced, only the first word is used for hex output.

Example:



6.3.3.2 REAL CONVERSION

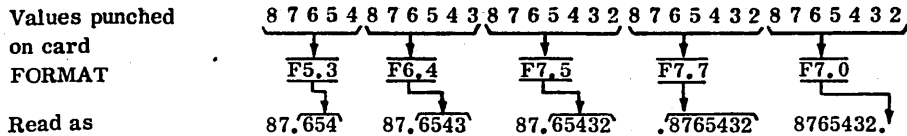
The specifications for real conversion are Fw.d and Ew.d.

Fw.d INPUT

The field descriptor Fw.d indicates that the external field occupies w positions, the fractional part of which consists of d digits. The field is scanned from left to right and embedded blanks are interpreted as zeros.

The basic input field consists of an optional sign followed by a string of digits which may contain a decimal point. If the decimal point is present, it overrides the d specification of the field descriptor.

Example:



Fw.d OUTPUT

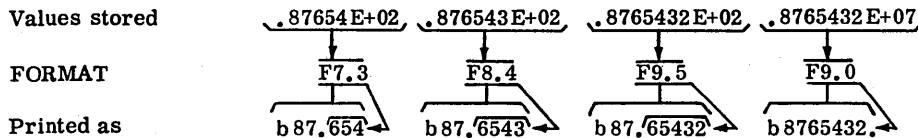
The basic output field occupies w positions. The corresponding list element must be a floating-point quantity which appears as a decimal number, right-justified in field w with possible leading blanks, as

$$\Delta s x_1, \dots, x_n$$

- Where: Δ is a possible blank fill
- s is the sign; minus if the number is negative, blank or omitted if the number is positive
- x_1, \dots, x_n is a string of digits containing a decimal point. The number of digits to the right of the decimal point is specified by d in the $Fw.d$. If d is zero, the digits to the right of the decimal point do not appear. d may contain a maximum of 19 digits.

If the field is too short to accommodate the number, asterisks fill the output field. If the field w is longer than required, the number is right-justified with blank fill to the left.

Example:



Ew.d INPUT

The number in the input field w is converted to a floating-point number and stored. The total number of characters in the input field is specified by w . The field is scanned from left to right and embedded blanks are interpreted as zeros.

The basic input field consists of an optional sign followed by a string of digits which may contain a decimal point. The basic field may be followed by an exponent of one of the following forms:

- Signed integer constant
- E followed by an integer constant
- E followed by a signed integer constant

The value of the exponent must not exceed ± 39 after normalization of the input field. The normalized number is the mantissa and the characteristic.

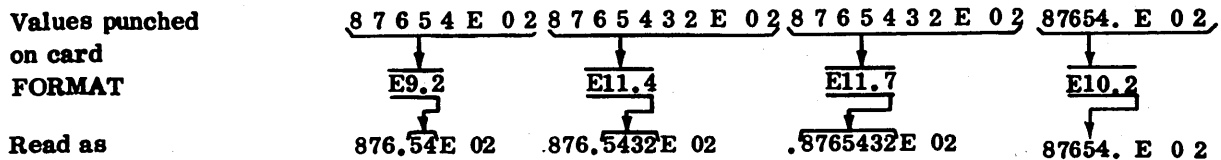
Permissible combinations:

+1.327E-04	Integer, fraction, and exponent
-32.721	Integer and fraction
+328E+5	Integer and exponent
.629E-1	Fraction and exponent
+136	Integer only
.0762	Fraction only

Normalized as	<u>Mantissa</u>	<u>Characteristic</u>
	.1327	E - 03
	-.32721	E + 02
	.328	E + 08
	.629	E - 01
	.136	E + 03
	.762	E - 01

A decimal point in the input number always overrides d. The field length specified by w in Ew.d should always be the same as the length of the input field containing the input number. When it is not, incorrect numbers may be read, converted, and stored. The field w includes the significant digits, signs, decimal point, E, and exponent.

Example:



Ew.d OUTPUT

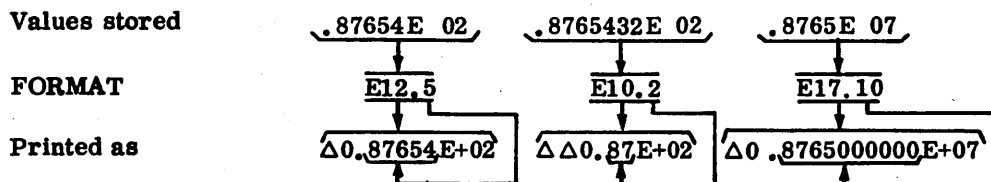
For output, floating-point numbers in storage are converted to the FORTRAN character form. The field occupies w positions in the output record; the corresponding floating-point number appears right-justified in the field as

$$\Delta s 0 . x_1, \dots, x_n E \pm ee \quad 0 \leq ee < 39$$

- Where:
- Δ is a possible blank fill
 - s is the optional sign; minus if negative, blank or suppressed if positive
 - x_1, \dots, x_n are the n most significant rounded digits of the value of the output data
 - ee are the digits of the exponent

Field w must be long enough to contain the specified number of digits, signs, decimal point, and exponent. For E conversion, w must be greater than or equal to d+7. The maximum number of digits in d is 19. If field w is too small to contain the output value, asterisks fill the field. If the field is longer than the output value, the number is right-justified with blank fill to the left.

Example:



6.3.3.3 DOUBLE-PRECISION CONVERSION

The specification for double-precision conversion is Dw.d.

Dw.d INPUT

The number in the input field w is converted to a double-precision floating-point number and stored. The total number of characters in the input field is specified by w. The field is scanned from left to right and embedded blanks are interpreted as zeros.

The basic input field consists of an optional sign followed by a string of digits which may contain a decimal point. The basic field may be followed by an exponent of one of the following forms:

- Signed integer constant
- D (or E) followed by an integer constant
- D (or E) followed by a signed integer constant

The value of the exponent must not exceed ± 39 after normalization of the input field. The normalized number is the mantissa and the characteristic.

Permissible combinations:

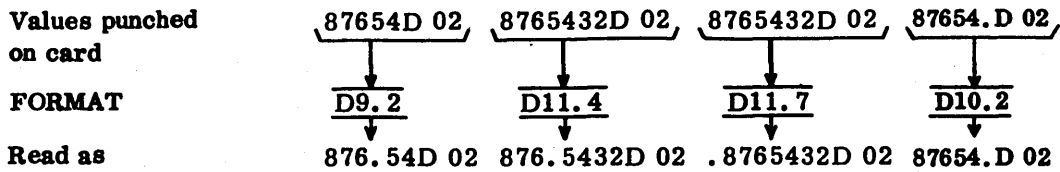
+1.327D-04	Integer, fraction, and exponent
-32.721	Integer and fraction
+328D+5	Integer and exponent
.629D-1	Fraction and exponent
+136	Integer only
.0762	Fraction only

Normalized as	<u>Mantissa</u>	<u>Characteristic</u>
	.1327	D - 03
	-.32721	D + 02
	.328	D + 08
	.629	D - 01
	.136	D + 03
	.762	D - 01

A decimal point in the input number always overrides d. The field length specified by w in Dw.d should always be the same as the length of the input field containing the input number. When it is not,

incorrect numbers may be read, converted, and stored. The field w includes the significant digits, signs, decimal point, D, and exponent.

Example:



Dw, d OUTPUT

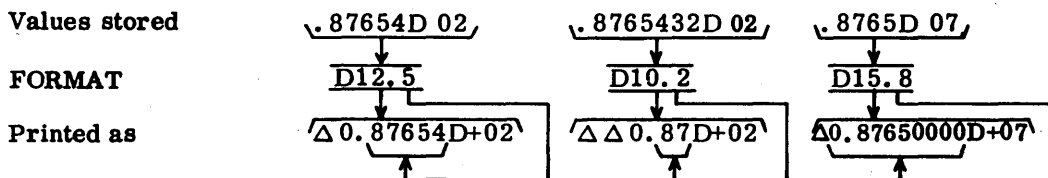
For output, double-precision floating-point numbers in storage are converted to the FORTRAN character form. The field occupies w positions in the output record; the corresponding double precision floating point number appears right-justified in the field as:

$$\Delta s 0 . x_1, \dots, x_n D \pm dd \quad 0 \leq dd < 39$$

Where: Δ is a possible blank fill
 s is the sign; minus if negative, blank or suppressed if positive
 x_1, \dots, x_n are the n most significant rounded digits of the value on the output data
 dd are the digits of the exponent

Field w must be long enough to contain the specified number of digits, signs, decimal point, and exponent. For D conversion, w must be greater than or equal to $d+7$. The maximum number of digits in d is 19. If the field w is too small to contain the output value, asterisks fill the field. If the field is longer than the output value, the number is right-justified with blank fill to the left.

Example:



6.3.4 ALPHANUMERIC CONVERSION

Aw and Rw specify I/O of alphanumeric data. The internal representation is ASCII. Refer to Appendix H for ASCII code.

Aw I/O

On input, the Aw specification accepts as list elements any two Hollerith characters. If the field width w is two or more, the rightmost two characters from the external input field are stored as the list element. If w equals one, the character from the external input field is left-justified in storage with a trailing blank. (27)

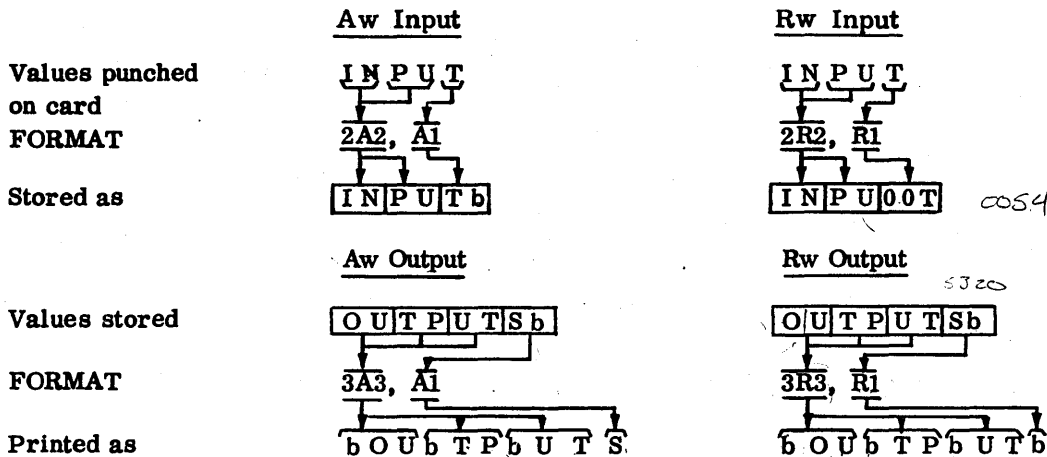
The A conversion outputs w Hollerith characters from a two-character list element. If w is two or more, the two characters from memory appear right-justified in the external output field preceded by blanks. If w equals one, the leftmost character from memory is stored in the output field.

Rw I/O

This specification is the same as the Aw specification with the following exceptions.

- On input, if w equals one, the character taken from the external input field is right-justified in storage with a leading hexadecimal 00.
- On output, if w equals one, the rightmost character from memory is stored in the output field.

Examples:



6.3.5 EDITING SPECIFICATIONS

Editing specifications are used to provide alphanumeric headings and comments, define spacing between characters and lines, skip records, and begin new records.

nH INPUT

On input, the H specification is used to place Hollerith characters in a pre-existing format. The n is an unsigned integer specifying the number of characters to the right of H that are to be placed in the format.

Unlike the A and R specifications, the characters input by an H specification are not stored in memory to be referenced by a symbolic name; instead, they are placed in a FORMAT statement already established in the source program. A READ instruction referencing this FORMAT statement will obtain a set of characters from an input device, such as a punched card, and place them in the FORMAT specification, replacing characters previously established.

Example:

The source program contains the instruction

```

READ (1, 15)

15 FORMAT (22HREPLACE THIS STATEMENT)
    
```

When this instruction is executed with the following input card

```

┌ DETERMINATION OF SIGMA
    
```

the heading DETERMINATION OF SIGMA is placed into FORMAT 15, replacing the character set REPLACE THIS STATEMENT.

Subsequently, the output statement

```

WRITE (3, 15)
    
```

would produce the printed line

```

DETERMINATION OF SIGMA.
    
```

The number of characters in the H input must exactly equal the number of characters pre-established in the FORMAT. If necessary, blanks can be used to balance out the input. It is immaterial what characters appear in the original source program FORMAT. In the preceding example, the instruction could be written

```

READ (1, 15)

15 FORMAT (22H^^^^^^^^^^^^^^^^^^^^^^^^)
    
```

The value of the H input specification lies in the flexibility it gives in varying a FORMAT specification at time of execution.

nH OUTPUT

This specification provides for the output of any set of Hollerith characters, including blanks, in the form of comments, titles, and headings. n is an unsigned integer specifying the number of characters to the right of H that will be transmitted to the output record. H denotes a Hollerith field.

Examples:

Source program

```
WRITE(3,20)
20 FORMAT(28H BLANKS COUNT IN AN H FIELD.)
```

produces output record

```
BLANKS COUNT IN AN H FIELD.
```

Source program

```
A = 1.5
WRITE(3,30)A
30 FORMAT(6H LMAX=, F5.2)
```

produces output record

```
LMAX= 1.50
```

LITERAL FREE-FIELD I/O

The literal free-field descriptor causes Hollerith information to be read into or written from the characters specified between two delimiters. The delimiters may be asterisks or single quotes. If the delimiters are asterisks, then embedded asterisks are not allowed. If the delimiters are single quotes, then embedded single quotes are not allowed.

Example:

```
WRITE (3,20)
20 FORMAT (* THIS IS A FREE FIELD FORMAT*)
or 20 FORMAT (' THIS IS A FREE FIELD FORMAT')
```

produces the output record

```
THIS IS A FREE FIELD FORMAT
```

6.3.6 NEW RECORD SPECIFICATIONS

A slash, signalling the end of an ASCII record, may appear anywhere in a FORMAT statement. It need not be separated by commas. A slash at the end of a FORMAT causes a record to be skipped, since the end of the list itself signals the end of a record. Likewise, a slash at the beginning of a FORMAT skips a record since the initiation of the list is itself a new record. Multiple slashes can be used to skip a number of records; however, the repeat specification does not apply to the slash. N slashes in the middle of a FORMAT skips N-1 records since the first slash merely signals a new record. N slashes at the beginning or end of a FORMAT list skips N records.

Examples:

These examples refer to the reading of records. They apply equally to output statements.

FORMAT(I2/F10.5)	Reads two records in succession
FORMAT(I2//F10.5)	Reads first record, skips second record, and reads the third record
FORMAT(I2///F10.5)	Reads one record, skips three, and reads the fifth
FORMAT(I2, F10.5/)	Reads one record and skips one record
FORMAT(I2/F10.5//)	Reads one record with I2, reads a second record with F10.5, and then skips two records
FORMAT(///F10.5)	Skips three records and reads a fourth with F10.5

6.3.7 BLANK FIELD SPECIFICATION

The general form of this specification is nX, where n is the number of blank spaces to be skipped on input or the number of blanks to be inserted in output.

Examples:

nX INPUT

The following values are to be read from a card:

X = 0.54321

Y = 3.25

I = 4321

The input statements

READ(1,20)X, Y, I

20 FORMAT(F10.5, 5X, F5.2, 10X, I10)

will interpret the input card as follows:

1	10	11	15	16	20	21	30	31	40
	54321	5X	325	10X					4321

nX OUTPUT

The following values are to be printed out.

```
IOTA      = 7
ALPHA     = 13.6
BETA      = 1462.37
```

The output statements

```
WRITE(3,44)IOTA, ALPHA, BETA
44 FORMAT(I2,6X,F6.2,7X,E12.5)
```

prints out

I2	F6.2	7X	E12.5
7	13.60		0.14623E+04

6.3.8 REPEATED FORMAT SPECIFICATIONS

Any format specification may be repeated by using a positive integer repetition constant r as follows:

r (spec)

Where: spec is any conversion specification except nX or nH.

Example:

```
WRITE (3,10) I, K, A, B, C
10 FORMAT (I2, I2, F8.4, F8.4, F8.4)
```

could be written

```
WRITE (3,10) I, K, A, B, C
10 FORMAT (2I2, 3F8.4)
```

Only one level of group repeat is allowed; group repeats may not be nested.

When the format control reaches the last outer right parenthesis of the specification, a test is made to determine if the I/O list is exhausted. If it is, control terminates. If another list element is specified, control returns to the group repeat specification terminated by the last preceding outer right parenthesis. If no repeat specification exists, control returns to the first left parenthesis of the specification.

Examples:

READ(1,10)N1, N2, A1, A2, M1, M2, B1
10 FORMAT (2I2, 2F6.2)

First card	N1	N2	A1	A2
Second card	M1	M2	B1	

READ(1,11)N1, N2, A1, A2, B1, B2
11 FORMAT (2I2, (2F6.2))

First card	N1	N2	A1	A2
Second card		B1	B2	

READ(1,12)N1, N2, A1, A2, B1, B2
12 FORMAT(2I2, 2(F6.2))

First card	N1	N2	A1	A2
Second card		B1		
Third card		B2		

READ(1,13)N1, N2, A1, A2, M1, M2, L1, L2
13 FORMAT((2I2), 2F6.2)

First card	N1	N2	A1	A2
Second card	M1	M2		
Third card	L1	L2		

6.3.9 FORMAT SPECIFICATION IN ARRAYS

The formatted READ and WRITE statements may contain an array name in place of the reference to a FORMAT statement label. When an array is referenced in such a manner, the first part of the information contained in the array, taken in the natural order, must constitute a valid format specification. There are restrictions on the information contained in the array following the right parenthesis that ends the format specification. The format specification which is to be inserted in the array has the same form as that defined for a FORMAT statement; that is, it begins with a left parenthesis and ends with a right parenthesis. The format specification may be inserted in the array by use of a READ statement together with the A format or by use of a DATA statement. If the ANSI option is used, the integer array containing the format must be typed SINGLE.

Example:

```
DIMENSION IFMT (40)
READ (1, 20) (IFMT (1), I=1, 40)
20 FORMAT (40A2)
READ (1, IFMT) A
```

Source data

```
(1H1, 5X, 'OBJECT ^ TIME ^ FORMATTING'/6X, F6.2)
```



A FORTRAN program consists of a main program with or without auxiliary procedures and subprograms. Auxiliary sets of statements are used to evaluate frequently used mathematical functions, to perform repetitious calculations, and to supply data specifications and initial values to the main program. MS FORTRAN provides six procedures and subprograms:

- Statement function
- Intrinsic function
- Basic external function
- External function
- External subroutine
- Block data subprogram

The intrinsic function and the basic external function are furnished with the system. They are used to evaluate standard mathematical functions. The others are user-defined. The statement function and intrinsic function are compiled within the main program, the basic external function is furnished with the system, and the others are compiled separately. The first five are referred to as procedures, since each is an executable unit that performs its set of calculations when referenced. The first four are called functions; they return a single result to the point of reference. The last three are subprograms; they are user-defined and are compiled independently. The block data subprogram supplies specifications and initial values to labeled common. Table 7-1 outlines these categorical divisions.

Use of procedures and subprograms is determined by their individual capabilities. If the program requires the evaluation of a standard mathematical function, then an intrinsic function or a basic external function is used (Tables 7-3 and 7-4). If a single nonstandard computation is needed repeatedly, a statement function is inserted in the program. If a number of calculations are required to obtain a single result, a function subprogram is written; if a number of calculations are required to obtain an array of values, a subroutine is written. When the program requires initial values in labeled common, a BLOCK DATA subprogram is used.

Table 7-1. Subdivision of Procedures and Subprograms

STATEMENT FUNCTION	INTRINSIC FUNCTION	BASIC EXTERNAL FUNCTION	EXTERNAL FUNCTION	EXTERNAL SUBROUTINE	BLOCK DATA SUBPROGRAM
User-defined	Compiler-defined		User-defined		
Compiled within the referencing program	Not compiled — LIBRARY —		Compiled externally to the referencing program		
PROCEDURE: Any defined calculation that can be referenced and which will exchange values between reference and definition through a list of arguments.					
		EXTERNAL PROCEDURE: A procedure defined externally to the program unit that references it.			
FUNCTION: A procedure that supplies a single result to be used at the point of reference.					
		EXTERNAL FUNCTION: A function defined externally to the program unit that references it.			
			SUBPROGRAM: A user-defined set of statements compiled independently of the program unit which references it or to which it supplies specifications and initial values.		
			PROCEDURE SUBPROGRAM: An external procedure that is defined by FORTRAN statements.		SPECIFICATION SUBPROGRAM: A subprogram without reference that supplies specifications and initial values to labeled common.

7.1 ARGUMENTS

7.1.1 ACTUAL

Procedures exchange values with referencing programs through argument lists. Arguments in the list of a referencing program are called actual arguments since they represent actual values relative to the referencing program.

7.1.2 DUMMY

Arguments listed in the procedure definition are called dummy arguments since they serve to exchange values between the reference list and the procedure calculations. Because of one-to-one correspondence, actual arguments and dummy arguments must agree in order, number, and type. The argument name cannot be the same as the subprogram name. A list of arguments that can be used with functions and subroutines is given in Table 7-2.

7.2 STATEMENT FUNCTION

A statement function is defined by a single statement in the program unit in which it is referenced. It must precede the first executable statement of the program unit and follow the specification statements, if any. During compilation, the statement function definition is compiled once at the beginning of the program; a transfer to this definition is generated whenever the statement function reference appears as an operand in an expression.

The format is:

$$f(a_1, a_2, \dots, a_n) = e$$

Where: **f** is the symbolic name of the function
 a is a dummy argument (at least one must be included)
 e is a defining arithmetic expression

The statement function is referenced by the appearance of its symbolic name followed by a list of actual arguments in an arithmetical or logical expression. Execution of the statement function calculation returns a single value to the reference.

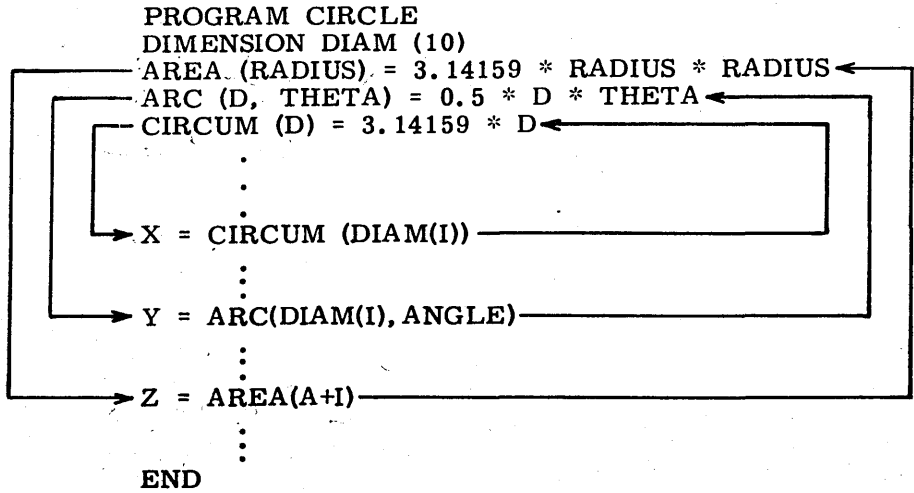
Example:

The following program calculates various parameters of a set of circles (one to ten). Input is an array of diameters (DIAM). The calculations include the determination of area, arc length, and circumference. These are given by statement functions at the beginning of the program which are referenced as needed.

Table 7-2. Permissible Arguments for Functions and Subprograms

	ACTUAL ARGUMENTS	FORMAL ARGUMENTS
STATEMENT FUNCTION	Constant Variable Array element Arithmetic expression	Variable
INTRINSIC FUNCTION	(Refer to Table 7-3)	
BASIC EXTERNAL FUNCTION	(Refer to Table 7-4)	
EXTERNAL FUNCTION	Constant Variable Array element Array name Any expression The name of an external procedure A BYTE or SIGNED BYTE variable if	Variable Array name External procedure name (May not appear in COMMON or DATA statement)
EXTERNAL SUBROUTINE	<ol style="list-style-type: none"> 1. It is passed as an integer (16 bit) variable or array, and 2. It is specified as BYTE or SIGNED BYTE in the FUNCTION or SUBROUTINE definition. <p><u>For subroutine only:</u> The name of the current procedure.</p>	

Note: Actual arguments and their corresponding formal arguments must agree in order, type, and number.



Explanation: The first reference is contained in the statement:

```
X=CIRCUM(DIAM(I))
```

in which the subscript I has been determined by calculations in the program. This reference places the actual argument DIAM(I) in the statement function:

```
CIRCUM(D)=3.14159*D
```

via the dummy argument D. The calculation is made and a single value for CIRCUM is returned to the referencing statement. The next reference supplies two actual arguments, DIAM(I) and ANGLE, to the statement function for ARC through the dummy arguments D and THETA. A single value for ARC is returned to the referencing statement.

The third reference uses an arithmetic expression, A+I, for an actual argument. This enters the statement function calculation for AREA through the dummy argument RADIUS. A single value for AREA is returned to the referencing statement.

7.3 SUPPLIED FUNCTIONS

To evaluate frequently used mathematical functions, MS FORTRAN supplies predefined calculations as well as references to library routines contained in the system. The predefined calculations are called intrinsic functions, and the references to the library routines are called basic external functions.

The intrinsic function inserts simple sets of calculations into the object program at compile time. The basic external function deals with more complex evaluations by inserting a reference to a library routine in the object program. The names of the supplied functions, their data types, and permissible arguments are predefined (Tables 7-3 and 7-4). References using these functions must adhere to the format defined in the tables. The type of a supplied function cannot be changed by a type statement.

Table 7-3. Intrinsic Functions

INTRINSIC FUNCTIONS	DEFINITION	NUMBER OF ARGUMENTS	SYMBOLIC NAME	TYPE OF ARGUMENT	TYPE OF FUNCTION
Absolute value	$ a $	1	ABS IABS DABS	Real Integer Double	Real Integer Double
Float	Conversion from integer to floating point	1	FLOAT DFLT	Integer Integer	Real Double
Fix	Conversion from floating point to integer	1	IFIX DFIX	Real Double	Integer Integer
Transfer of sign	Sign of a_2 times $ a_1 $ The sign of 0 is +.	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Obtain most significant part of double-precision argument		1	SNGL	Double	Real
Express single-precision argument in double-precision form		1	DBLE	Real	Double
Logical sum	Form the bit by bit logical sum of a_1 and a_2 .	2	OR	Integer	Integer
Exclusive OR	Complement those bits of a_1 which are one in a_2 .	2	EOR	Integer	Integer
Logical product	Form the bit by bit logical product of a_1 and a_2 .	2	AND	Integer	Integer
Complement	Complement a_1	1	NOT	Integer	Integer

7.3.1 INTRINSIC FUNCTION

An intrinsic function is a compiler-defined set of calculations that is inserted in the referencing program at compile time. If the set involves only a few machine instructions, it is inserted in the program every time the reference appears. This method is called in-line code. The intrinsic functions IABS, OR, EOR, AND, and NOT produce in-line code. If the set of instructions needed to evaluate the intrinsic function is lengthy, the compiler identifies it and generates the calling sequence to this set of calculations whenever the function is referenced.

The intrinsic function is referenced by the appearance of the function name with appropriate arguments in an arithmetic or logical statement. A list of intrinsic functions is given in Table 7-3. The name of an intrinsic function listed in this table must satisfy all of the following requirements:

1. The name must not appear in an EXTERNAL or a RELATIVE statement (Sections 7.4.3 and 7.4.4), an array name or an array element, or be the name of a statement function (Section 7.2).
2. The name must not appear in a type statement (Section 6.1.4) declaring it to be other than the type specified in the table.
3. Every appearance of the name must be followed by a list of arguments enclosed in parentheses, unless the name is in a type statement.

The use of an intrinsic function in one program unit precludes the use of its name as the name of a different entity in another program unit in that same program. If a user-defined subprogram has the same name as an intrinsic function, the name of the subprogram must be further defined by a type declaration (Section 6.1.4) or by an EXTERNAL statement (Section 7.4.3).

7.3.2 BASIC EXTERNAL FUNCTION

A basic external function is a call on one of the predefined library routines included with the system. These library routines are used to evaluate standard mathematical functions such as sine, cosine, square root, etc. When a reference to a basic external function appears in an expression, the compiler identifies it and generates the calling sequence in the object program. A basic external function is referenced by the appearance of the function name with appropriate arguments in an arithmetic or logical statement. A list of basic external functions is given in Table 7-4.

NOTE

The compiler does not generate the calculations for a basic external function; it generates the call to the library routine for that particular function in the object program. At execution time, illegal values input to the basic external functions in Table 7-4 will give predetermined results (Table 7-5).

Table 7-4. Basic External Functions

BASIC EXTERNAL FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	SYMBOLIC NAME	TYPE OF ARGUMENT	TYPE OF FUNCTION
Exponential	e^a	1	EXP DEXP	Real Double	Real Double
Natural logarithm	$\log(a)$	1	ALOG DLOG	Real Double	Real Double
Trigonometric sine	$\sin(a)$	1	SIN DSIN	Real Double	Real Double
Trigonometric cosine	$\cos(a)$	1	COS DCOS	Real Double	Real Double
Hyperbolic tangent	$\tanh(a)$	1	TANH	Real	Real
Square root	$(a)^{1/2}$	1	SQRT DSQRT	Real Double	Real Double
Arctangent	$\arctan(a)$	1	ATAN DATAN	Real Double	Real Double
End of file check on unit a	EOF(a) Check previous read on unit a for end-of-file. 2 is returned if none. 1 is returned if EOF.	1	EOF	Integer	Integer
Floating-point fault	IFALT(a) If a is 0, overflow is tested. If a is 1, divide fault is tested. If a is 2, underflow is tested. A 2 is returned if the condition has not occurred, a 1 otherwise.	1	IFALT	Integer	Integer
Parity error check on unit	IOCK(a) Check previous read or write on unit a for parity error. 2 is returned if none. 1 is returned if parity error occurred.	1	IOCK	Integer	Integer

Table 7-5. Basic External Functions, Predetermined Results

BASIC EXTERNAL FUNCTION	ARGUMENT VALUE	RESULT
SIN or COS	$ Z > 2^{20}$	0
EXP	$ Z > 87.0$	∞
ALOG	$Z \leq 0$	∞
SQRT	$Z < 0$	$-\sqrt{ Z }$
DSIN or DCOS	$ Z > 2^{20}$	0
DEXP	$ Z > 87.0$	∞
DLOG	$Z \leq 0$	∞
DSQRT	$Z < 0$	$-\sqrt{ Z }$

7.4 SUBPROGRAMS

Subprograms are used to implement programming capability beyond the limitations of supplied functions and the statement function. Although written as a subset of another program, the subprogram is compiled separately; it has its own independent variables, and its use is not limited to communication with the program for which it was written. Procedure subprograms handle routine calculations unique to the user; specification subprograms are used to enter values into labeled COMMON and to supply such program information as is given by DIMENSION, DATA, EQUIVALENCE, and COMMON statements.

Procedure subprograms may be function or subroutine. In both cases, a series of FORTRAN statements is used to perform a calculation in conjunction with another program that calls it into operation. Subprograms are called either by the appearance of the name in an arithmetic or logical statement (function subprogram) or by a CALL statement (subroutine subprogram). Distinctive features of procedure subprograms include the ability to pass array names and external procedure names as arguments (Section 7.4.3). A BYTE or SIGNED BYTE may become an argument in a reference to a subprogram. The user is reminded that only the address of the BYTE or SIGNED BYTE is passed to the subprogram.

A subprogram returns control to a calling program through one or more RETURN statements or by an assigned GO TO statement, whose assign variable has been defined by an ASSIGN statement in the subprogram or in the calling program. If the ASSIGN statement is in the calling program, the assign variable must be passed as an actual argument or be in common.

Because they are independent programs, procedure subprograms must terminate with an END statement to signal to the compiler that the physical end of the source program has been reached. An END statement causes a return to the calling program and may replace a final RETURN statement.

Formal arguments specified in SUBROUTINE or FUNCTION statements may be referenced in EQUIVALENCE statements and BYTE statements.

Example:

```
INTEGER FUNCTION TEST6 (A,B)
INTEGER A(10), B(10)
BYTE (IA2, A(2) (15=13))
EQUIVALENCE (M, B(3))
N = M + IA2
TEST6 = N - IA2
END
```

The fundamental differences between a function and subroutine subprogram are given in Table 7-6.

There is one type of specification subprogram, the block data subprogram.

Table 7-6. Differences Between Function and Subroutine Subprograms

FUNCTION	SUBROUTINE
Passes a value back to the calling statement	Does not pass a value back to the calling statement
Referenced by the name appearing in an arithmetic or logical statement	Referenced by a CALL statement
Must have one or more arguments	Need not have any arguments
Name is typed by first letter or by the type designation appearing before the word FUNCTION	No type associated with name

7.4.1 FUNCTION SUBPROGRAM

A function subprogram is a collection of FORTRAN statements headed by a FUNCTION statement and written as a separate program to perform a set of calculations when its name appears in an arithmetic or logical expression in the referencing program.

The format is

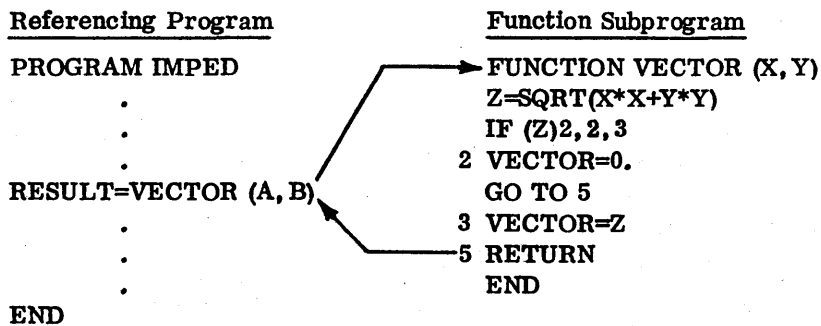
t FUNCTION f(a₁, a₂, ..., a_n)

Where: t is the type designation: INTEGER, REAL, DOUBLE PRECISION, or empty
f is the symbolic name of the function to be defined
a₁ are dummy arguments which may be variable names, array names, or external procedure names. They cannot be the same as the function name.

The function subprogram accepts arguments from the referencing program through the argument list and through common. It returns a single value through the function name. The function name must be assigned a value by appearing at least once in the subprogram as a variable on the left side of an arithmetic statement or by appearing in the list of an input statement.

When a function reference is encountered in an expression, control transfers to the function subprogram indicated. When RETURN or END is encountered in the function subprogram, control returns to the statement containing the function reference, or an assigned GO TO statement transfers control to an indicated statement.

Example:



The function subprogram is referenced by the appearance of the name and list in the statement

RESULT=VECTOR (A, B)

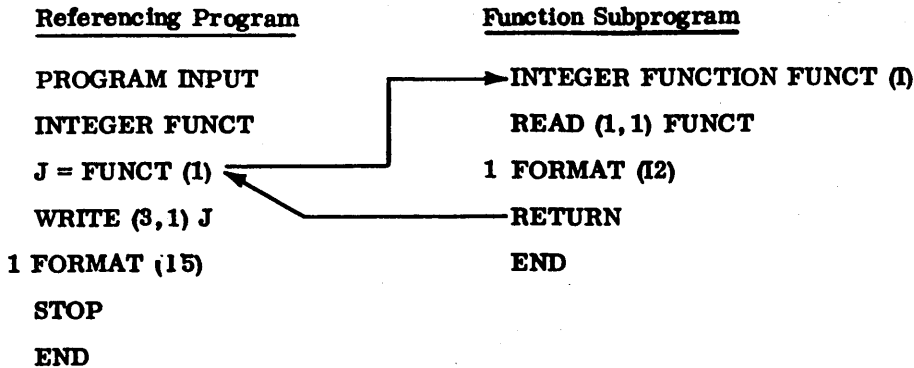
The values represented by the actual arguments A and B are communicated to the subprogram through the dummy arguments X and Y.

The function subprogram can also return results through its arguments and/or through common.

The first calculation in the subprogram involves the appearance of a secondary reference: SQRT. This reference passes the calculated value in the parentheses to the basic external function for obtaining a square root. The result is returned to the subprogram and placed in storage location Z. Z is then tested to see if it is positive. If not, function name VECTOR is equated to zero and that value is returned to the reference; if it is positive, function name VECTOR is equated to that positive value and returned to the reference.

The following example shows how a function subprogram can establish a value for the function name by using an input statement rather than an arithmetic statement.

Example:



Since the subprogram is intended to deal with integer values and its name is implicitly real, the name is typed integer in the referencing program and in the FUNCTION statement of the subprogram. The subprogram is referenced by the statement

J = FUNCT (1)

which arbitrarily passes the constant 1 as an actual argument. It enters the subprogram through dummy argument I in the FUNCTION statement but is never used. This step is performed solely to satisfy the requirements of a function subprogram. The subprogram reads in the value from a card and stores it in the location designated by the name of the function subprogram, where it is available to the referencing program which stores it in J and then prints it out.

7.4.2 SUBROUTINE SUBPROGRAM

A subroutine subprogram is a collection of FORTRAN statements headed by a SUBROUTINE statement and written as a separate program to perform a set of calculations when called by a referencing program.

The formats are:

```
SUBROUTINE s
SUBROUTINE s(a1, a2, ..., an)
```

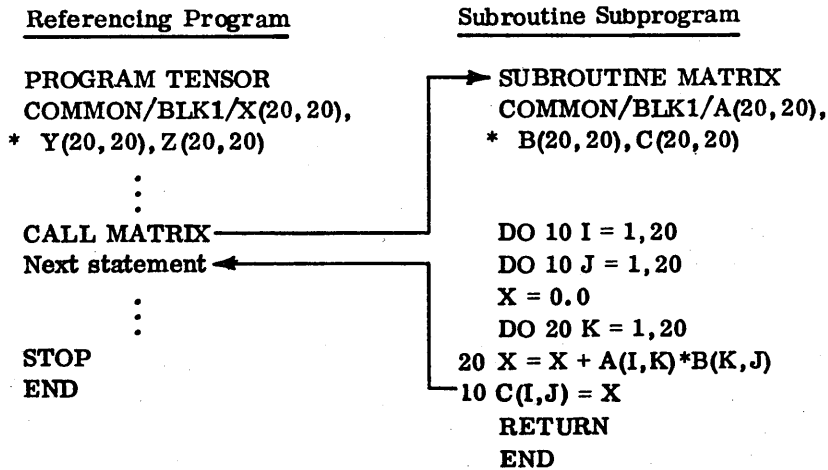
Where: s is the symbolic name of the subroutine to be defined

a₁ - a_n are dummy arguments which may be variable names, array names, or external procedure names. They cannot be the same as the subroutine name.

A CALL statement transfers control from the calling program to the subroutine. A RETURN or END statement returns control to the next executable statement following the CALL statement in the referencing program, or an assigned GO TO statement transfers control to an indicated statement.

The subroutine subprogram accepts arguments from the calling program and/or through common. It can return one or more results through its arguments and/or through common.

Example:



The referencing program reserves storage for three successive arrays in labeled common. It is assumed that two of these arrays, X and Y, have values stored in them before the CALL statement is reached. The CALL statement transfers control to the subroutine without passing any arguments. The subroutine performs the matrix multiplication of the first two arrays and stores the results in the third. Control is returned to the next statement after the CALL in the referencing program. The subroutine obtains the values for its calculations from the labeled common block and returns the results it derives to the same labeled common block.

7.4.3 EXTERNAL STATEMENT

The name of an external procedure (basic external function, function subprogram, or subroutine subprogram) can be passed as an argument to a procedure subprogram (function or subroutine) provided that name has been first declared in an EXTERNAL statement.

The format is:

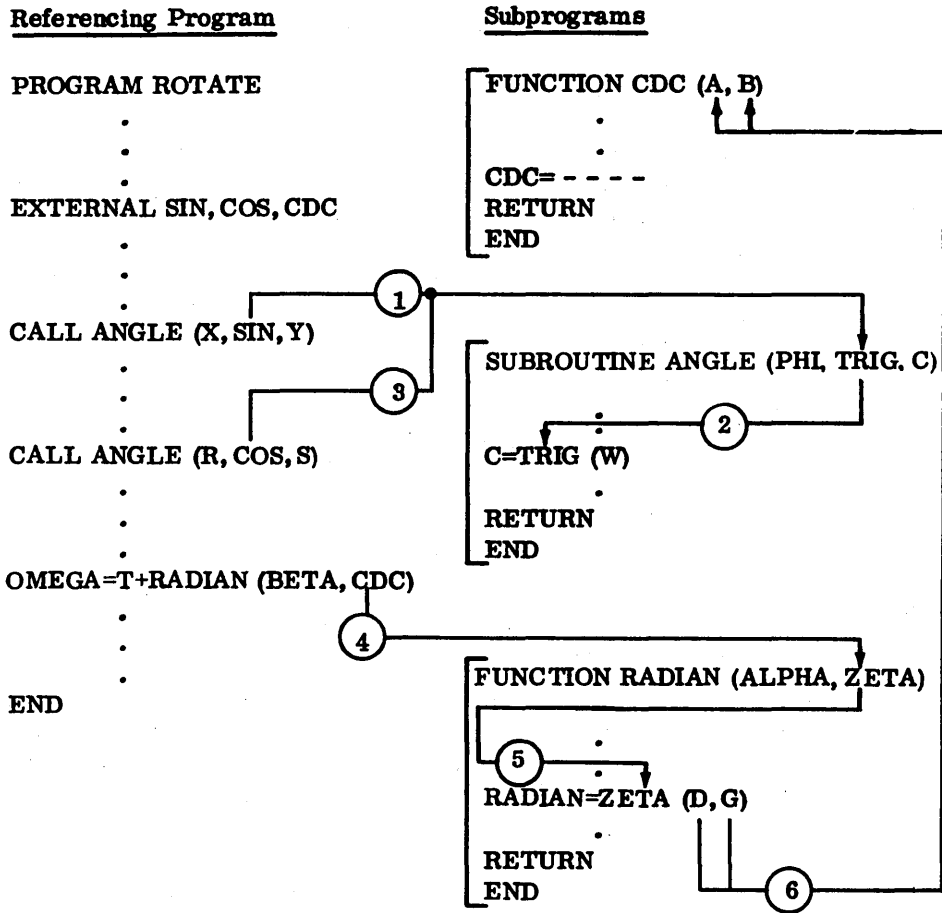
```
EXTERNAL v1, v2, ..., vn
```

Where: v_i is an external procedure

Use of this statement enables the compiler to distinguish the address of an external procedure from that of an ordinary variable.

Once the name of the external procedure is passed through a dummy variable of the subprogram, it operates as a procedure in the subprogram just as it would in the calling program.

Examples:



The referencing program declares three external procedures in the EXTERNAL statement. SIN and COS are basic external functions in the system library and CDC is a user-written function subprogram. This declaration makes it possible to pass these names as arguments to subprograms where they can operate as procedures and evaluate variables in the subprograms.

The first subprogram reference in PROGRAM ROTATE is

CALL ANGLE (X, SIN, Y)

The name SIN is passed to the dummy argument TRIG in SUBROUTINE ANGLE.

The appearance of the name TRIG in the statement

C = TRIG (W)

makes that statement the equivalent of

$$C = \text{SIN } (W)$$

and the basic external function SIN is called into operation using the argument W.

The next reference is a call to the same subroutine

CALL ANGLE (R,COS,S)

This time the name COS is passed through the dummy argument TRIG and the statement

$$C = \text{TRIG } (W)$$

becomes the equivalent of

$$C = \text{COS } (W)$$

which calls into operation the basic external function for COS using the argument W.

The final reference is the appearance of the external function name RADIAN in the statement

$$\text{OMEGA} = T + \text{RADIAN } (\text{BETA}, \text{CDC})$$

The name CDC is passed through the dummy argument ZETA in the subprogram FUNCTION RADIAN.

The appearance of ZETA in the statement

$$\text{RADIAN} = \text{ZETA } (D,G)$$

makes it the equivalent of

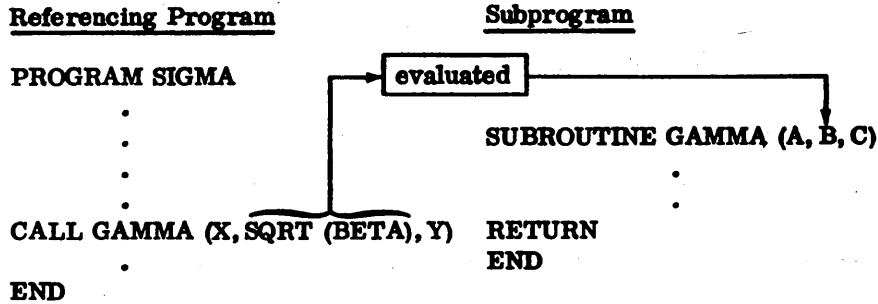
$$\text{RADIAN} = \text{CDC } (D,G)$$

which references the user-written subprogram FUNCTION CDC.

The actual arguments D and G are passed through the dummy arguments A and B. This subprogram calculates a value for CDC and returns it to the radian subprogram. From here it is returned to the referencing program.

The following example illustrates a use of a function reference as an argument which does not require declaration in an EXTERNAL statement.

Example:



In the CALL statement, SQRT is not itself an argument; the function SQRT (BETA) is evaluated first and the result is passed as an argument to the dummy variable B in the subroutine. Thus, SQRT need not be declared in an EXTERNAL statement.

7.4.4 RELATIVE STATEMENT

The RELATIVE statement declares a name to be an external procedure name.

The format is:

RELATIVE $v_1, v_2, v_3, \dots, v_n$

Where: v_1 is an external procedure name

Appearance of a name in a RELATIVE statement declares that name to be an external procedure name. When the run-anywhere option has been selected, appearance of a name in a RELATIVE statement causes all references to this procedure to be made in a way that preserves the run-anywhere characteristic. An external procedure name that is to be passed as an actual argument to a procedure subprogram cannot appear in a RELATIVE statement. (It would appear in an EXTERNAL statement.)

7.4.5 CALL STATEMENT

Subroutines are referenced by the appearance of a CALL statement in the referencing program.

The formats are:

CALL s
CALL s (a_1, a_2, \dots, a_n)

Where: s is the name of the subroutine being called
a is an actual argument

The name may not appear in any specification statement in the calling program except in an EXTERNAL or a RELATIVE statement.

The CALL statement transfers control to the subroutine named. When a RETURN or END statement is encountered in the subroutine, control returns to the next executable statement following the CALL in the referencing program. If the CALL statement is the last statement in a DO loop, looping continues until the DO is satisfied.

7.4.6 RETURN STATEMENT

This statement marks the logical end of a procedure subprogram; it returns control to the calling program.

7.4.7 BLOCK DATA SUBPROGRAM

Initial values can be entered into the elements of the labeled common block at load time with the block data subprogram. This is a nonexecutable subprogram composed of specification statements, a DATA statement, and an END statement.

The first statement of this subprogram must be

BLOCK DATA

It is followed by the specification statements:

COMMON
EQUIVALENCE
DIMENSION

Type statements

These specification statements are followed by the DATA statements, which enter initial values into one or more elements of labeled common. If an element in a common block is being given an initial value, specification statements for the entire block must be included. Elements in unlabeled common may not be given initial values by the block data subprogram.

Example:

```
BLOCK DATA
COMMON/ENTER/A, C, D, I, K
DIMENSION A(4), B(4), C(5), D(2), I(3), J(3), K(2)
EQUIVALENCE (A, B), (I, J)
DATA A(1), A(2), A(3), A(4)/1.1, 2.2, 3.3, 4.4/, C(1), C(2), C(3), C(4), C(5)/
*1.1, 2.2, 3.3, 4.4, 5.5/, D(1), D(2)/10.1, 10.2/, I(1), I(2), I(3), K(1), K(2)/
*1, 2, 3, 4, 5/
END
```

Explanation:

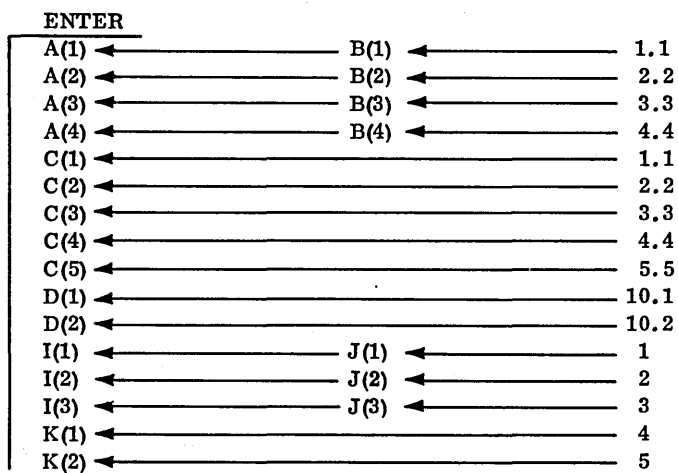
The DIMENSION statement reserves storage for the following arrays.

A(1)	B(1)	C(1)	D(1)	I(1)	J(1)	K(1)
A(2)	B(2)	C(2)	D(2)	I(2)	J(2)	K(2)
A(3)	B(3)	C(3)		I(3)	J(3)	
A(4)	B(4)	C(4)				
		C(5)				

The COMMON statement enters arrays A, C, D, I, K, in that order in the common block labeled ENTER.

The EQUIVALENCE statement enters arrays B and J into the labeled common block to share storage with arrays A and I.

The DATA statement enters the following values into the designated locations of the labeled common block:



8.1 COMPILATION

The user provides the source programs. Mass Storage FORTRAN continues compiling source programs until it encounters a statement of the following form:

ΔMON

MON must be in character positions 2, 3, and 4 immediately preceded by a blank (Δ) in character position 1. This statement must immediately follow the END statement which marks the physical end of a source program unit. The MON statement returns control to the operating system.

The OPT statement allows the user to select options from the standard input device. The selected options may exist in three ways:

1. L, X, P, C options assumed with omission of OPT card.
2. OPT card with desired options after column 5.
3. No options specified by OPT card. This permits options to be entered through the standard input comment device.

OPT must be in character positions 2, 3, and 4 immediately preceded by a blank in column 1. Options must be preceded by a blank in column 5. The options may begin any column after column 5.

The options are:

- P Relocatable binary object program output on standard binary output device.
- L Source program listing (contains the generated statement numbers) on the standard list device.
- A Object code listing on the list device.
- M Condensed object code listing on the list device. Listing contains generated statement numbers and first word of object code generated by each statement.
- R Run-anywhere object code. This option allows a program to be executed anywhere in allocatable core.

CAUTION

Programs compiled with the R option do not execute properly in partitioned core or at addresses above 8000₁₆. In addition, programs compiled with the R option which call user-written subroutines must not declare formal parameters located in Part 1. For example:

```
PROGRAM
:
:
EXTERNAL IPART1
:
:
CALL SUB (PART1)
:
:
END
```

does not execute properly if the R option is used and IPART1 is in Part 1 memory.

- K ANSI FORTRAN compatibility; integers occupy two computer words
- X Relocatable binary object program placed on the load-and-go file. Disk or drum is used for load-and-go.
- O Optional compilation. If this option is used, statements containing an O in column 1 are compiled as if column 1 were blank. If the option is not used, statements with an O in column 1 are treated as comment cards. This option allows optional prints, dumps, pauses, etc., while debugging.
- V Variable number of parameters in subroutine/function calls. If this option is used:
 1. Error 13 is not given if the number of arguments in a subroutine/function call differs from the number of arguments in the first reference to the subroutine/function.
 2. The return jump to the subroutine/function is preceded by an ENA N where N is the number of arguments being passed.
 3. Both the subroutine and its calling program must be compiled using the V option.
 4. The subroutine may determine the number of arguments in its call by referencing the special variable N9PARS.
- C Cross reference. An alphabetized listing of constants, variables, externals, and labels, including the address of the symbol, its type, and a list of statement references.

Unrecognized parameters and blanks are ignored. Compiler diagnostics are provided on the list device regardless of the options selected. Compilation error diagnostics are in Appendix L. A fatal diagnostic prevents generation of any object code.

The following examples illustrate the output from the various options for a small test program.

OPTION L

Note that full compilation is not done. Only a statement syntax check is made.

```

1          PROGRAM FTNOPT
           C
           S          EXAMPLE FOR FORTRAN OPTIONS
           C
2          DIMENSION A(5),I(5)
3          DO 1 II=1,5
4          I(II)=II*3/A(II)
5          1 CONTINUE
6          CALL SUBEXM(A,I)
           O          CALL SUBEXM(A,I,II,J)
7          J=K*6*C
8          IF(FUNEXM(4,9)) 10,20,10
9          10 GOTO 20
10         20 CONTINUE
11        END

```

OPTIONS LA

```

1          PROGRAM FTNOPT
           C
           S          EXAMPLE FOR FORTRAN OPTIONS
           C
2          DIMENSION A(5),I(5)
3          DO 1 II=1,5
4          I(II)=II*3/A(II)
5          1 CONTINUE
6          CALL SUBEXM(A,I)
           O          CALL SUBEXM(A,I,II,J)
7          J=K*6*C
8          IF(FUNEXM(4,9)) 10,20,10
9          10 GOTO 20
10         20 CONTINUE
11        END

```

0000	0000		NAM	FTNOPT
0000	1819	FTNOPT	JMP*	.00001
0001	000A	A	BSS	10
0008	0005	I	BSS	5
0010	0001	II	RSS	1
0011	0003	0003\$	NUM	3
0012	0001	J	RSS	1
0013	0001	K	BSS	1
0014	0006	0006\$	NUM	6

	0015	0002	C	RSS	2	
	0017	0004	0004\$	NUM	4	
	0018	0009	0009\$	NUM	9	
3	0019	0A01	.00001	ENA	1	
	001A	68F5		STA*	II	
4	001B	C8F4	.00004	LDA*	II	
	001C	28F4		MUI*	0003\$	
	001D	6829		STA*	.00005	
	001E	5400		RTJ*	FLOAT	
	001F	7FFF				
	0020	0046	P	ADC	.00005	
	0021	5400		RTJ*	HFLOT	
	0022	7FFF				
	0023	FA14		NUM	-1515	
	0024	0010	P	ADC	II	
	0025	7FFE	P	ADC	A	-2
	0026	000A	P	ADC	I	-1
5	0027	D8E8	1	RAO*	II	
	0028	0A05		ENA	5	
	0029	98E6		SUB*	II	
	002A	0131		SAM	1	
	002B	18EF		JMP*	.00004	
6	002C	5400		RTJ*	SUREXM	
	002D	7FFF				
	002E	0001	P	ADC	A	
	002F	0008	P	ADC	I	
7	0030	5CEE		RTJ*	(FLOAT)	
	0031	0014	P	ADC	0006\$	
	0032	5CEF		RTJ*	(HFLOT)	
	0033	9D40		NUM	-25279	
	0034	0015	P	ADC	C	
	0035	0047	P	ADC	.00006	
	0036	5CE8		RTJ*	(FLOAT)	
	0037	0013	P	ADC	K	
	0038	5CE9		RTJ*	(HFLOT)	
8	0039	E140		NUM	-7871	
	003A	0047	P	ADC	.00006	
	003B	0012	P	ADC	J	
	003C	5400		RTJ*	FUNEXM	
	003D	7FFF				
	003E	0017	P	ADC	0004\$	
	003F	0018	P	ADC	0009\$	
	0040	5CE1		RTJ*	(HFLOT)	
	0041	D400		NUM	-11263	
	0042	0047	P	ADC	.00006	
	0043	C804		LDA*	.00006	
	0044	0104		SAZ	4	
9	0045	1804	10	JMP*	20	
	0046	0001	.00005	BSS	1	
	0047	0002	.00006	BSS	2	
11	0049	5400	20	RTJ*	QBSTP	
	004A	7FFF				
11	0000	0000		END	0	

PROGRAM LENGTH \$004B (75)

EXTERNALS

HFLOT QBSTP FLOAT SUBEXM FUNEXM

OPTIONS LM

Note condensed object code listing. This form is useful when the list device is a teletype.

```
1          PROGRAM FTNOPT
           C
           $
           C          EXAMPLE FOR FORTRAN OPTIONS
2          DIMENSION A(5),I(5)
3          DO 1 II=1,5
4          I(II)=II*3/A(II)
5          1 CONTINUE
6          CALL SUBEXM(A,I)
           0 CALL SUBEXM(A,I,II,J
7          J=K+6*C
8          IF(FUNEXM(4,9)) 10,20,10
9          10 GOTO 20
10         20 CONTINUE
11        END
```

3	0019	0A01	.00001	ENA	1
4	001B	C8F4	.00004	LDA*	II
5	0027	D8E8	1	RAO*	II
6	002C	5400		RTJ*	SUBEXM
7	0030	5CEE		RTJ*	(FLOAT)
8	0039	E140		NUM	-7871
9	0045	1804	10	JMP*	20
11	0049	5400	20	RTJ*	QBSTP
11	0000	0000		END	0

PROGRAM LENGTH 50048 (75)

EXTERNALS

HFL0T QBSTP FLOAT SUBEXM FUNEXM

OPTIONS LAR

Note that no program relocatable addresses are generated; hence, the program can run in allocatable core.

```

1          PROGRAM FTNOPT
           C
           S          EXAMPLE FOR FORTRAN OPTIONS
           C
2          DIMENSION A(5),I(5)
3          DO 1 II=1,5
4          I(II)=II*3/A(II)
5          1 CONTINUE
6          CALL SUBEXM(A,I)
           O          CALL SUBEXM(A,I,II,J)
7          J=K+6*C
8          IF(F(INEXM(4,9))) 10,20,10
9          10 GOT0 20
10         20 CONTINUE
11        END
    
```

	0000	0000	NAM	FTNOPT
			.00001	
	0000	1819	FTNOPT JMP*	.00002
	0001	000A	A BSS	10
	0008	0005	I BSS	5
	0010	0001	II BSS	1
	0011	0003	0003\$ NUM	3
	0012	0001	J BSS	1
	0013	0001	K BSS	1
	0014	0006	0006\$ NUM	6
	0015	0002	C BSS	2
	0017	0004	0004\$ NUM	4
	0018	0009	0009\$ NUM	9
	0019	5802	.00002 RTJ*	.00005
	001A	FFE5	ADC	.00001
	001B	0001	.00005 BSS	1
	001C	C8FE	LDA*	.00005
	001D	88FC	ADD*	.00005
	001E	68FC	STA*	.00005
3	001F	0A01	ENA	1
	0020	68EF	STA*	II
4	0021	C8EE	.00006 LDA*	II
	0022	28EE	MUI*	0003\$
	0023	682A	STA*	.00007
	0024	5400	RTJ+	FLOAT
	0025	7FFF		
	0026	8027	ADC	.00007
	0027	5400	RTJ+	HFLOAT
	0028	7FFF		
	0029	5FA1	NUM	24481

-1

	002A	FFES		ADC	II	
	002B	FFD3		ADC	A	-2
	002C	FFD0		ADC	I	-1
	002D	4000		NUM	16384	
5	002E	D8E1	1	RAO*	II	
	002F	0A05		ENA	5	
	0030	98DF		SUB*	II	
	0031	0131		SAM	1	
	0032	18EE		JMP*	.00006	
6	0033	5400		RTJ*	SUBEXM	
	0034	7FFF				
	0035	FFC0		ADC	A	
	0036	FFD4		ADC	I	
7	0037	5CED		RTJ*	(FLOAT)	
	0038	FFD0		ADC	0006\$	
	0039	5CEE		RTJ*	(HFLOT)	
	003A	59D4		NUM	22996	
	003B	FFD9		ADC	C	
	003C	0012		ADC	.00008	
	003D	5CE7		RTJ*	(FLOAT)	
	003E	FFD4		ADC	K	
	003F	5CE8		RTJ*	(HFLOT)	
8	0040	5E14		NUM	24084	
	0041	000D		ADC	.00008	
	0042	FFCF		ADC	J	
	0043	5400		RTJ*	FUNEXM	
	0044	7FFF				
	0045	FFD1		ADC	0004\$	
	0046	FFD1		ADC	0009\$	
	0047	5CE0		RTJ*	(HFLOT)	
	0048	5D40		NUM	23872	
	0049	0005		ADC	.00008	
	004A	C804		LDA*	.00008	
	004B	0104		SAZ	4	
9	004C	1804	10	JMP*	20	
	004D	0001	.00007	BSS	1	
	004E	0002	.00008	BSS	2	
11	0050	5400	20	RTJ*	QBSTP	
	0051	7FFF				
11	0000	0000		END	0	

PROGRAM LENGTH \$0052 (82)

EXTERNALS
HFLOT QBSTP FLOAT SUBEXM FUNEXM

OPTIONS LAK

This form allocates two words of memory for each integer. The actual executable code only uses one of the two words.

```

1      PROGRAM FTNOPT
      C
      S      EXAMPLE FOR FORTRAN OPTIONS
      C
2      DIMENSION A(5),I(5)
3      DO 1 II=1,5
4      I(II)=II*3/A(II)
5      1 CONTINUE
6      CALL SUBEXM(A,I)
      0      CALL SUBEXM(A,I,II,J)
7      J=K+6*C
8      IF(FUNEXM(4,9)) 10,20,10
9      10 GOTO 20
10     20 CONTINUE
11     END

```

	Address	Hex	Symbol	Type	Value	
	0000	0000	NAM	FTNOPT		
	0000	1821	FTNOPT	JMP*	.00001	
	0001	000A	A	BSS	10	
	0008	000A	I	BSS	10	
	0015	0002	II	BSS	2	
	0017	0003	0003\$	NUM	3	
	0018	0002	J	BSS	2	
	001A	0002	K	BSS	2	
	001C	0006	0006\$	NUM	6	
	001D	0002	C	BSS	2	
	001F	0004	0004\$	NUM	4	
	0020	0009	0009\$	NUM	9	
3	0021	0A01	.00001	ENA	1	
	0022	68F2		STA*	II	
4	0023	0A02	.00004	ENA	2	
	0024	28F0		MUI*	II	
	0025	682E		STA*	.00005	
	0026	C8EE		LDA*	II	
	0027	28EF		MUI*	0003\$	
	0028	682C		STA*	.00006	
	0029	5400		RTJ+	FLOAT	
	002A	7FFF				
	002B	0054 P		ADC	.00006	
	002C	5400		RTJ+	HFL0T	
	002D	7FFF				
	002E	FAF1		NUM	-1294	
	002F	0015 P		ADC	II	
	0030	7FFE P		ADC	A	-2
	0031	0053 P		ADC	.00005	
	0032	0009 P		ADC	I	-2
	0033	4000		NUM	16384	

5	0034	D8E0	1	RAO*	II	
	0035	0A05		ENA		5
	0036	98DE		SUB*	II	
	0037	0131		SAM		1
	0038	18EA		JMP*	.00004	
6	0039	5400		RTJ+	SUBEXM	
	003A	7FFF				
	003B	0001	P	ADC	A	
	003C	0008	P	ADC	I	
7	003D	5CEC		RTJ*	(FLOAT)	
	003E	001C	P	ADC	00065	
	003F	5CED		RTJ*	(HFLOT)	
	0040	9D40		NUM	-25279	
	0041	001D	P	ADC	C	
	0042	0055	P	ADC	.00007	
	0043	5CE6		RTJ*	(FLOAT)	
	0044	001A	P	ADC	K	
	0045	5CE7		RTJ*	(HFLOT)	
8	0046	E140		NUM	-7871	
	0047	0055	P	ADC	.00007	
	0048	0018	P	ADC	J	
	0049	5400		RTJ+	FUNEXM	
	004A	7FFF				
	004H	001F	P	ADC	00045	
	004C	0020	P	ADC	00095	
	004D	5CDF		RTJ*	(HFLOT)	
	004E	D400		NUM	-11263	
	004F	0055	P	ADC	.00007	
	0050	C805		LDA*	.00007	
	0051	0105		SAZ		5
9	0052	1805	10	JMP*	20	
	0053	0001	.00005	RSS		1
	0054	0001	.00006	RSS		1
	0055	0002	.00007	RSS		2
11	0057	5400	20	RTJ+	08STP	
	0058	7FFF				
11	0000	0000		END		0

PROGRAM LENGTH \$0059 (89)

EXTERNALS
HFLOT 08STP FLOAT SUBEXM FUNEXM

OPTIONS LX

Note that the full compilation has taken place.

```
1          PROGRAM FTNOPT
           C
           C      EXAMPLE FOR FORTRAN COMPILER OPTIONS
           C
2          DIMENSION A(5),I(5)
3          DO 1 II=1,5
4          I(II)=II*3/A(II)
5          1 CONTINUE
6          CALL SUBEXM(A,I)
7          J=K*6*C
8          IF(FUNEXM(4,9)) 10,20,10
9          10 GO TO 20
10         20 CONTINUE
11        END
```

PROGRAM LENGTH 5004F (79)

OPTS = LX

EXTERNALS

QBQFIX FLCT QBSTP FLOAT SUBEXM FUNEXM

OPTIONS PX

Note that no listing output is generated, but full compilation has occurred with object and load and go output.

OPTS = PX

OPTIONS LOVC

This form allows compilation of optional statements that have an O in column 1. The V option allows subroutine calls to have varied numbers of parameters defined in the call statement.

```
1          PROGRAM FTNOPT
           C
           C      EXAMPLE FOR FORTRAN COMPILER OPTIONS
           C
2          DIMENSION A(5),I(5)
3          DO 1 II=1,5
4          I(II)=II*3/A(II)
5          1 CONTINUE
6          PAUSE 0001
7          CALL SUBEXM(A,I)
8          J=K*6*C
9          IF(FUNEXM(4,9)) 10,20,10
10         10 GO TO 20
11         20 CONTINUE
12        END
```

OPTIONS LC

This option instructs the compiler to produce a cross reference of constants, variables, externals, and labeled statements.

```
1          PROGRAM FTNOPT
           C
           $          EXAMPLE FOR FORTRAN OPTIONS
           C
2          DIMENSION A(5),I(5)
3          DO 1 II=1,5
4          I(II)=II*3/A(II)
5          1 CONTINUE
6          CALL SUBEXM(A,I)
           0          CALL SUBEXM(A,I,II,J)
7          J=K+6*C
8          IF(FUNEXM(4,9)) 10,20,10
9          10 GOTO 20
10         20 CONTINUE
11        END
```

PROGRAM LENGTH \$004B (75)

EXTERNALS
HFLOT QBSTP FLOAT SUBEXM FUNEXM

***** LIST OF SYMBOLS *****

CONSTANTS :

VALUE	ADDRESS	REFERENCED BY STATEMENT NR :
0003 (3)	0011	4
0004 (4)	0017	7
0006 (6)	0014	7
0009 (9)	0018	7

VARIABLES :

NAME	TYPE	ADDRESS	REFERENCED BY STATEMENT NR :
A	REAL	0001	1,4,6
C	REAL	0015	7
I	INTEGER	0008	1,4,6
II	INTEGER	0010	1,4
J	INTEGER	0012	6,7
K	INTEGER	0013	7

EXTERNALS :

NAME	TYPE	ADDRESS	REFERENCED BY STATEMENT NB :
FLOAT	REAL.FN.	001F	
FUNEXM	REAL.FN.	003D	7
HFLOT	REAL.FN.	0022	
QBSTP	INTEGER.FN.	004A	
SUBEXM	SUBROUTINE	002D	5

LABELED STATEMENTS :

LABEL	ADDRESS	REFERENCED BY STATEMENT NB :
1	0027	1,5
10	0045	7,8,9
20	0049	7,9,10
FTNOPT	0000	1

8.2 EXECUTION

When option P is selected, a punched output is generated containing the binary object program. This output may be loaded by MSOS. This form may also be loaded by the system initializer. When option X is selected, the binary object program is output as a load-and-go file on disk or drum. It can be loaded and executed in the same run as the compilation.

Upon completion of the load, any unsatisfied external references in the object program are satisfied from the program library.

Execution time error messages are listed in Appendix M.

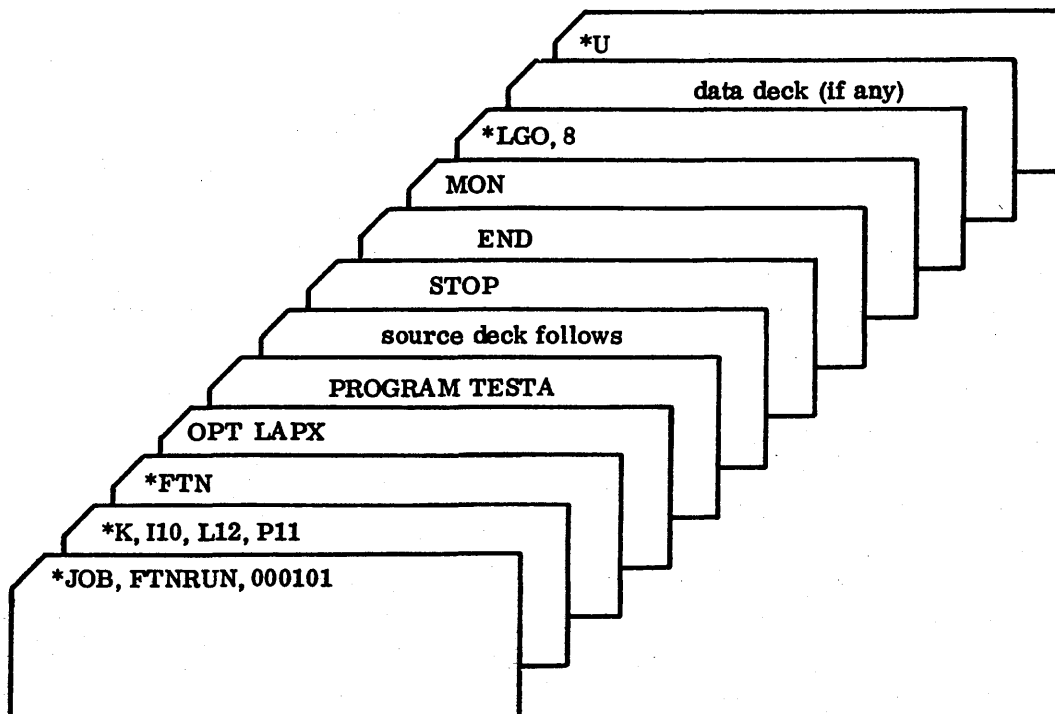
8.3 PROGRAM OPERATING PROCEDURES

This section outlines the method of compiling and executing a FORTRAN program under MSOS. To illustrate the step-by-step interaction between the operator and the system, typical values are selected for the parameters. It is assumed the system is without a timer.

The following logical unit designations are made.

<u>Device</u>	<u>Unit No.</u>
Card punch	11
Card reader	10
Mass storage device	8
Printer	12

The FORTRAN deck is placed in the card reader with the system control cards around it as shown in the following illustration:



The JOB card is utilized in the job processor to begin a new background job. The next card assigns the I/O units. In this example, it specifies standard input from the card reader, list output to the printer, and binary output to the card punch. The *FTN card calls in the compiler to compile the source deck. The OPT card is read by the compiler, in this case list, list assembly code, punch relocatable binary, and put binary on the load and go file are the options selected. The compiler then reads in the source deck and compiles the program. The MON card after the source deck releases the compiler and returns control to the job processor. The *LGO card instructs the job processor to load the object code for the program, along with any object library routines necessary to execute the program. It is assumed in this example that the program reads in the data deck during execution. After execution, control is returned to the job processor which reads in the *U card and returns control to the teletype.



This chapter discusses the use of the re-entrant ENCODE/DECODE and non-re-entrant ENCODE/DECODE run-time packages. These packages have, in general, reduced capability from the FORTRAN I/O run-time discussed in other chapters with an extension in the interface capability to MSOS monitor requests. The features throughout this section are to be used with one word integer-type variables wherever integer-type variables are used.

The re-entrant and non-re-entrant packages have an identical user interface. This duplication of capability allows initial program debugging in the background using the non-re-entrant version with a transfer to the re-entrant version for execution in the foreground.

The intrinsic functions defined in Table 7-3 and the basic external functions defined in Table 7-4 are also operable with the ENCODE/DECODE run-times. Two exceptions are the EOF and IOCK functions which can not be called from foreground.

9.1 RE-ENTRANT FORTRAN

Two characteristics of FORTRAN programs which execute in a multiprogramming environment are:

- Priority levels can be assigned to the different programs executing in the computer.
- The monitor and standard FORTRAN library are re-entrant.

9.1.1 PRIORITIES

Assigning different priorities to the programs in memory permits the monitor (the basic portion of MSOS which allocates the use of the computer on a priority basis) to determine the order in which programs execute. When a program asks the monitor to initiate an I/O request, control may be given to another program to execute, rather than waiting for completion of the I/O request. Upon completion of the request, if the completion priority is higher than the current executing program, control returns to the program which made the I/O request. The program currently executing is interrupted, and the monitor retains all pertinent information at the point of interruption. When control is eventually returned to this program of lower priority, all pertinent information saved upon interruption is restored. If the completion priority of the I/O request is not higher than the currently executing program, the completion of the I/O request is processed at a later time according to its priority. The process can be cascaded to the depth allowed by the monitor. In the standard release system levels 4, 5, and 6 are defined as re-entrant FORTRAN levels.

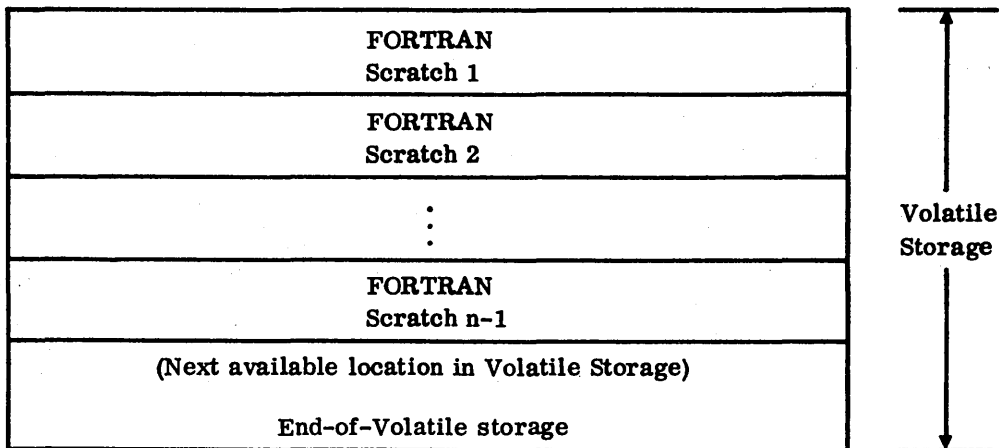
9.1.2 RE-ENTRANCY

A program which can be interrupted and re-entered by another program of high priority level is called re-entrant. Re-entrant programs require all pertinent information be saved upon interruption and restored when execution is resumed.

All programs or subprograms that may run at more than one level concurrently must be re-entrant. The FORTRAN library falls into the re-entrant category since it can be called from more than one priority level.

9.1.3 FORTRAN LIBRARY

All routines in the FORTRAN library use a scratch area in the communications region of the monitor (locations \$C5 to \$E5) for intermediate results. Interruption of a FORTRAN program by another FORTRAN program requires storing and restoring this scratch area into and from volatile storage in the monitor. Thus, n levels of interrupts by FORTRAN programs result in n-1 copies of the scratch area in volatile storage.



It is not desirable for n to assume large values since larger core requirements for the monitor restrict the amount of core available for user programs. Limiting the priority levels of FORTRAN programs to three or four levels restricts the number of interrupts FORTRAN programs can have. This holds the requirements on volatile storage to a reasonable size.

9.1.4 FORTRAN READ/WRITE STATEMENT PROCESSOR

In order to implement the FORTRAN READ/WRITE statement as part of a re-entrant statement processor, a deviation from the ANSI standard FORTRAN specifications was made for the following reasons:

- The size of the input/output buffer to be reserved in the statement processor is dependent upon the largest message for input/output by any FORTRAN program.
- Since the statement processor is re-entrant, either the buffer is stored in volatile storage on interruption (again requiring a large amount of volatile) or interrupts are inhibited until the complete buffer is input/output. Since several milliseconds are required to inhibit interrupts, this method would defeat the purpose of a multiprogramming system.

To resolve the preceding objections, the FORTRAN READ/WRITE statement processor places the responsibility of providing an input/output buffer upon the FORTRAN programmer. Also, since control is not returned to the FORTRAN program until a READ/WRITE statement has been completely processed, there is no chance of the user destroying the message by attempting to do more READ/WRITE processing into his buffer. This negates the necessity of storing and restoring his buffer.

Re-entrancy places a further restriction on the READ/WRITE statement. The FORMAT statement may designate only one input record (80 card columns) per READ statement.

9.1.5 FORTRAN/MONITOR RUN-TIME INTERFACE (FORTRA)

The monitor has provisions to request a mass storage READ/WRITE or an unformatted READ/WRITE, schedule the execution of a new program, schedule the execution of a new program after a time increment has elapsed, release core after execution of the current program is completed, etc. For the FORTRAN programmer, however, communication with the monitor is only possible through the FORTRAN/monitor run-time package. This package has entry points which generate specific requests to the monitor when called by a FORTRAN program. Thus, when a CALL READ is made with the logical unit equal to mass storage and the mass storage addresses are provided, the run-time package generates the necessary calling sequence to the monitor, then makes the input request and returns to the user's program.

CALL READ, CALL WRITE, CALL FREAD, and CALL FWRITE, which are entry points to the FORTRAN/monitor run-time package, are direct requests to the monitor. The READ/WRITE FORTRAN statements are used specifically for reads or writes with a FORTRAN FORMAT statement.

9.1.6 ENCODE/DECODE

If the FORTRAN/monitor run-time interface is used to transfer the record, ENCODE/DECODE provides the programmer with the capability to convert ASCII characters to hexadecimal data (DECODE) or to convert hexadecimal data to ASCII characters (ENCODE).

With the FORTRAN/monitor run-time interface, ENCODE/DECODE, and the READ/WRITE statement processor, the FORTRAN programmer has the full capabilities of data input/output and has sufficient control over the problem to achieve correct results in a multiprogramming real-time environment.

9.1.7 RUN-ANYWHERE PROGRAMS

So that FORTRAN programs can execute properly in allocatable core, a run-anywhere option was added to the FORTRAN compiler, removing all absolute address references from the compiled program.

CAUTION

Users are warned that programs compiled with the run-anywhere (R) option will not execute properly in partitioned core or at addresses above \$8000.

9.2 FORMAT SPECIFICATIONS

Data transmission between storage and an external unit requires a call to an I/O routine (READ, WRITE, etc.) and may require a FORMAT statement. The I/O call specifies the input/output device, the process, and a list of data to be transmitted. No FORMAT statement is required to transmit binary information, and a direct call to an I/O routine may be made. With ASCII information, a FORMAT statement specifies the type of conversion to be made on the data before or after transmission.

9.2.1 FORMAT STATEMENT

The FORMAT statement contains the specifications relating to the internal/external structure of the corresponding data elements.

FORMAT(spec₁, ..., k(spec_m, ...), spec_n, ...)

Where: spec is a format specification

k is an optional repetition factor which must be an unsigned integer constant

FORMAT statements are nonexecutable and may appear anywhere in the program.

9.2.2 FORMAT CONVERSION

The data elements in I/O lists are converted from external to internal or from internal to external representations according to conversion and editing specifications in the FORMAT statement. The FORMAT statement may contain both conversion and editing specifications. The format conversion specifications are:

Ew.d	Floating-point conversion with exponent	} Limited to output specifications only
Dw.d	Double-precision floating-point with exponent	
Fw.d	Floating-point conversion without exponent	
Iw or Iw.d	Decimal integer conversion	
\$w or Zw	Hexadecimal integer conversion	
Aw	Alphanumeric conversion	
Rw	Alphanumeric conversion	

The format editing specifications are:

wX	Intra-line spacing
xH	Heading and labeling
Asterisk	*String of ASCII Characters*
or	
Quote	'String of ASCII Characters'
/	Line-feed/new record

Both w and d are unsigned integers. w specifies the field width (the number of character positions in the record) and d specifies the number of digits to the right of the decimal within the field.

9.2.3 CONVERSION SPECIFICATIONS

Dw,d OUTPUT

This specification converts double-precision floating-point numbers in storage to ASCII characters, including an exponent for output. The field occupies w positions in the output record with d digits as

the most significant part of the fraction. The corresponding floating-point number will appear right-justified in the field as:

$\pm .\text{xxxxxxD}\pm\text{ee}$

Where: $0 \leq \text{ee} \leq 39$

Let: A contain -1276.45 or .001276450D0

And: FORMAT(D15.4)

Result: ^^^^-.1276D+04 or
^^^^.1276D-02

Ew.d OUTPUT

This specification converts floating-point numbers in storage to ASCII characters, including an exponent for output. The field occupies w positions (minimum 6) in the output record with d digits ($\neq 0$) as the most significant part of the fraction. The corresponding floating-point number appears right-justified in the field as:

$\pm .\text{XXXXXXXX}\pm\text{ee}$

Where: $0 \leq \text{ee} \leq 39$

The fractional portion of the number contains a maximum of six digits. If the field width is too short to accommodate the number, an asterisk appears in the most significant position to indicate an error.

Let: A contain -67.32 or .06732

And: FORMAT(E10.3)

Result: ^-.673E 02 or
^^.673E-01

Let: A contain -67.32 or .06732

And: FORMAT(E7.3)

Result: *.6E 02 or
*.6E-01

Fw.d OUTPUT

This specification converts floating-point numbers in storage to ASCII characters, excluding an exponent for output. The field occupies w positions in the output record with d digits to the right of the decimal. The corresponding floating-point number appears right-justified in the field as:

$\pm\text{X}\dots\text{X}.\text{X}\dots\text{X}$.

The range of the internal number represented must be from 10^{-5} to $10^{+5}-1$. If this range is exceeded, the field is filled with asterisks and no error flag is returned as in Section 9.4.3. If the field width

is too short to accommodate the number, an asterisk appears in the most significant character position to indicate the error.

Let: A contain +32.694

And: FORMAT(F7.3)

Result: ^32.694

Let: A contain -32767.0

And: FORMAT(F7.3)

Result: *2767.0

Fw,d INPUT

This specification converts ASCII characters in storage to a floating-point number and scales the string of integer digits by 10^{-d} . The field occupies w positions in the input record; a decimal point in the input record causes the d portion of the conversion specifications to be ignored. With $d = 0$, both fields must be specified to indicate no scaling. The range of the internal number represented must be from 10^{-5} to $10^{+5}-1$.

Let: INPUT = A(1) = ^9.

A(2) = 35

Where: A contains ASCII characters

And: FORMAT(F4.2)

Result: 9.35

Let: INPUT = A(1) = ^-

A(2) = 52

A(3) = .3

And: FORMAT(F6.3)

Result: -52.3

Let: INPUT = A(1) = ^9

A(2) = .5

A(3) = 20

A(4) = ^-

A(5) = 50

A(6) = 60

And: FORMAT(2F6.2)

Result: 9.520 and

-50.36

Iw, Iw,d OUTPUT

This specification converts integer values to ASCII characters with 10^{-d} scaling if d is specified. The magnitude of the integer number must be from 10^{-5} to $10^{+5}-1$. If the field is wider than required, the

output quantity is right-justified and blank-filled. If the field width is too short, an asterisk appears in the most significant character position to indicate the error.

Let: N contain 301

And: FORMAT(I5)

Result: ^^301

Let: N contain -336

And: FORMAT(I5.3)

Result: -.336

Iw INPUT

This specification converts ASCII characters to an integer value. The magnitude of the number must be from $-(2^{+15}-1)$ to $2^{+15}-1$.

Let: INPUT = 1905

And: FORMAT(I4)

Result: N = 1905

Let: INPUT = 0,9,3,8,0,2

And: FORMAT(6I1)

Result: N(1) = 00 N(4) = 08
 N(2) = 09 N(5) = 00
 N(3) = 03 N(6) = 02

\$w or Zw OUTPUT

This specification converts a hexadecimal integer value in storage to ASCII characters for output. The field occupies w positions in the output record. If the field width is too short, an asterisk is inserted in the most significant character position. The magnitude of the internal number represented must be from $-(2^{+15}-1)$ to $2^{+15}-1$.

Let: N contain 03A2₁₆

And: FORMAT(\$6) or FORMAT (Z6)

Result: ^^3A2

Let: N contain 83A2₁₆

And: FORMAT(\$3) or FORMAT (Z3)

Result: *A2

\$w or Zw INPUT

This specification converts ASCII characters in storage to a hexadecimal integer value. The magnitude of the internal number represented must be from $-(2^{+15}-1)$ to $2^{+15}-1$.

Let: N contains AB in ASCII characters
And: FORMAT(\$2) or FORMAT (Z2)
Result: INPUT = 00AB

Aw OUTPUT

This specification is used to output ASCII characters. w characters/word are picked up, starting with the leftmost character, and stored in the output buffer. If the field width is greater than two, an error return occurs.

Let: N(1) = C^
N(2) = NX
N(3) = =^
N(4) = Y^
N(5) = 1^
And: FORMAT(A1,A2,3A1)
Result: CNX = Y1
Let: N(1) = CN
N(2) = 1=
N(3) = Y1
And: FORMAT(3A1)
Result: C1Y

Aw INPUT

This specification accepts as list elements any set of eight-bit characters including blanks. The internal representation is ASCII; the field width is w characters. If w exceeds two, an error return occurs. w characters are picked up as a left-justified ASCII word; the remaining spaces are blank filled.

Let: INPUT = CNXYYZ
And: FORMAT(6A1)
Result: N(1) = C^ N(4) = Y^
N(2) = N^ N(5) = Y^
N(3) = X^ N(6) = Z^
Let: INPUT = CNXYYZ
And: FORMAT(2A2, 2A1)
Result: N(1) = CN N(3) = Y^
N(2) = XY N(4) = Z^

Rw OUTPUT

This specification is the same as Aw specification except that the output quantity represents the rightmost quantity. If the field width is greater than one, an error return results.

Let: N(1) = 0A and N(2) = 0B
0 = 8 bits of zeros
And: FORMAT(2R1)
Result: AB

Rw INPUT

With this specification the input quantity goes to the designated storage location as a right-justified zero-filled word. If w is greater than one, an error return results.

Let: INPUT = AB
And: FORMAT(2R1)
Result: N(1) = 0A
N(2) = 0B
0 = 8 bits of zeros

9.2.4 EDITING SPECIFICATIONS

wX OUTPUT/INPUT

This specification may be used to include w blanks in an output record or to skip w characters on input to permit spacing of input/output quantities.

Let: A = -32.576
And: FORMAT(3X, F7.3)
Result: ^^^-32.576

wH OUTPUT/INPUT

This specification provides for the output of any set of eight-bit characters, including blanks in the form of comments, titles, and headings. w is an unsigned integer specifying the number of characters to the right of the H that are transmitted to the output record as ASCII characters. The H field may be used to read a new heading into an existing H field.

FORMAT(3X, 5HLABEL, 1X, 4HFOR, 6HOUTPUT)
Result: ^^LABEL^FOR^OUTPUT
FORMAT(1H1, 12H^NEW^HEADING)
Result: ^NEW^HEADING

QUOTE OR ASTERISK I/O

The asterisk field descriptor causes Hollerith information (excluding asterisks) to be read into or written from the characters specified between two asterisk delimiters. The single quote field descriptor causes Hollerith information (excluding single quotes) to be read into or written from the characters specified between two single quote delimiters.

Examples:

```
WRITE (3, 20)
20  FORMAT(*^THIS IS A HOLLERITH STRING*)
Result:  THIS IS A HOLLERITH STRING
WRITE (3, 30)
30  FORMAT('^THIS IS A HOLLERITH STRING')
Result:  THIS IS A HOLLERITH STRING
WRITE (3, 40)
40  FORMAT ('*', '**', ' THIS IS A HOLLERITH STRING', '**', '*')
```

NEW LINE

The slash, which signals the end of a line, may occur anywhere in the specification list. This generates a new line into the output record.

```
FORMAT(1X, 6HLINE^1, //7H^LINE^3)
RESULT  ^LINE^1
        ^LINE^3
```

9.2.5 SPECIAL CHARACTER SPECIFICATIONS

If a special character appears as the first character in the output record, the following interpretation is made:

```
1  Top-of-form
0  Line feed
```

```
FORMAT(1H1, 15X, 12HTOP-OF-FORM^)
```

The run-time converts the first character of the output buffer: if it is an ASCII code for 0 (\$ 30) to an ASCII code for a line feed (\$0D), and if it is an ASCII code for 1 (\$31) to an ASCII code for a top of form (\$0C). This technique does not require the use of the FORTRAN line printer logical unit to interpret form control.

9.2.6 REPEATED FORMAT SPECIFICATIONS

Any **FORMAT** specifications may be repeated by using an unsigned integer constant repetition factor (**k**) as follows:

k (spec)

Where: **spec** is any conversion specification. The level of repetitions is limited to one.

Thus, (**k₁(----k₂(----))**) is an error.

But (**k₁----**), **----k₂(----)**, **----k₃(----)**) does not result in an error.

9.3 FORTRAN READ/WRITE STATEMENT PROCESSOR

Input/output **FORTRAN** control statements (**READ/WRITE**) transfer information between core storage and external peripheral devices connected to the computer.

9.3.1 WRITE STATEMENT

WRITE(i, n) L

transfers information from storage locations given by identifiers in the list (**L**) to a specified peripheral device (**i**) according to the **FORMAT** statement (**n**).

```
        WRITE(10, 20) A, B, C
20     FORMAT(3F10. 6)
        WRITE(10, 30)
30     FORMAT(33H THIS STATEMENT HAS NO DATA LIST.)
```

9.3.2 READ STATEMENT

READ(i, n) L

transfers one record of information from a specified peripheral device (**i**) to storage locations named by the list (**L**) identifiers according to the **FORMAT** statement (**n**).

```
        READ(10, 20)X, Y, Z
20     FORMAT(3F10. 6)
        READ(10, 30)
30     FORMAT(33H(message)) where 33 blank spaces must appear between the H and the
        terminating parenthesis.
        READ(10, 40)(Z(K), K=1, 8)
40     FORMAT(F10. 4)
```

9.3.3 STATEMENT PROCESSOR

The statement processor (Q8QIO) serves as an interface between the FORTRAN READ/WRITE statement, the format processor (ENCODE/DECODE), and the input/output processor (Monitor READ/WRITE request processor). It allows the FORTRAN programmer to use the READ/WRITE statements as defined by FORTRAN with the following exceptions:

- The user must supply a buffer in which the format processing takes place.
- Eighteen temporary locations immediately preceding the buffer contain the calling sequence to the monitor for read/write processing and information for re-entrancy.
- Only one RECORD/READ statement on input may be executed; the FORMAT statement may specify only 80 columns of data for card input.
- The RECORD/WRITE statement on output may be as long as the space in the buffer allows with the following limitations: if the programmer has not specified a new line after 150 characters have been packed into the buffer, a carriage return is automatically inserted into the message and continues to be inserted every 150 characters until the FORMAT processing is complete.
- Noncompatible with ANSI FORTRAN option; no two-word integer values (K option).

9.3.4 CALL SETBFR

In order to communicate the starting location and the length of the user's buffer to Q8QIO, an entry point called SETBFR is provided. The call to transmit the information need only be made once and must precede any READ/WRITE statement. However, if the user's program makes a call to the dispatcher or a call to either ENCODE/DECODE, then a call to SETBFR must again be made prior to any READ/WRITE statement.

CALL SETBFR(buffer, length)

The first 18 words of the buffer contain the calling sequence for the I/O request and information for re-entrancy. The remainder contains the input/output message.

9.3.5 RESTRICTIONS

Length is the total length of the buffer which includes the 18 words needed by Q8QIO. This scratch area of 18 words has the following format:

- | | | |
|------|---|-----------------------------------|
| Word | 1 | Last word address (LWA) of buffer |
| | 2 | Request code for READ/WRITE |
| | 3 | Completion address |
| | 4 | Thread |
| | 5 | Logical unit |

- Word 6 Message length
- 7 First word address (FWA) of message
- 8 Sector address MSB (unformatted READ/WRITE)
- 9 Sector address LSB (unformatted READ/WRITE)
- 10 Q register of user
- 11 Return address of user's program
- 12 I register of user
- 13 READ/WRITE flag (ICODE)
- 14 LIST address
- 15 Total number of variables in LIST
- 16 ENCODE/DECODE—READ/WRITE flag (DEFLAG)
- 17 FORTRAN FORMAT flag
- 18 I register for restoring volatile
- 19 User's I/O message begins here

The greatest restriction on implementing the READ/WRITE statement processor was placed on the input side. This restriction limits each READ statement to request one input record or 80 columns of data for card input.

```

      READ(10, 20)(X(I, J), I=1, 10), J=1, 20)
20   FORMAT(10F8.4)

```

This example results in an error since the request specifies 20 cards of input. However, the following executes correctly:

```

      DO 30 J 1, 20
      READ(10, 20)(X(I, J), I=1, 10)
20   FORMAT(10F8.4)
30   CONTINUE

```

Unformatted READ/WRITE may be performed by use of the re-entrant READ/WRITE statement processor.

READ(i)L transfers one record of information directly from the device (i) into the storage locations named by the list (L) identifiers.

WRITE(i)L transfers information from the storage locations named by the list (L) identifiers to the device (i).

If the device is mass storage, words 8 and 9 of the buffer specified by CALL SETBFR must contain the sector address.

Unformatted READ/WRITE is not implemented in the non-re-entrant READ/WRITE statement processor.

9.3.6 FORMAT ERRORS

To determine if a format error occurred during processing of a READ/WRITE statement, the programmer may follow the READ/WRITE statement with a call to the function subroutine IOERR. An error is indicated if the function value is -1.

```
IF(IOERR(0).EQ.-1)GO TO 1000.
```

or IERROR=IOERR(0).
(and IERROR may be tested later)

9.3.7 I/O ERRORS

To determine if a hardware failure occurred during an I/O operation, the programmer may follow the READ/WRITE statement with a call to the function subroutine IRWERR:

```
IF(IRWERR(0).LT.0) GO TO 1000
```

or JERROR = IRWERR(0)
(and JERROR may be tested later)

The negative value of the function indicates that an I/O error occurred on the last READ/WRITE operation. This function is implemented only for the re-entrant ENCODE/DECODE run-time. Refer to the MSOS reference manual for a further description of I/O errors.

9.4 ENCODE/DECODE CALLS

The ENCODE/DECODE package gives the FORTRAN programmer the ability to transfer information under FORMAT specifications from one area of storage to another. For example, to transfer a floating-point number from a variable data list into an output buffer area with an F format specification, the programmer would use an ENCODE call to accomplish the conversion from floating-point representation to ASCII characters and pack the output buffer. ENCODE/DECODE functions use the ENCODE/DECODE run-time routines. Therefore, the formatting capabilities are as described in this chapter.

The parameters to an ENCODE/DECODE call are:

```
CALL ENCODE/DECODE (buffer, inform, n, list)
```

Where:

buffer is an area to ENCODE into or DECODE from; always contains information in ASCII form.

inform is an assigned variable when the statement label assigned is the statement number which represents the associated FORMAT statement.

n equals the number of variables to ENCODE/DECODE.

list equals the first word of the data list to input/output; always contains data in hexadecimal form.

9.4.1 ENCODE

ENCODE transmits *n* machine-language elements of the variable list according to *iform* into locations starting with the first word in buffer. Up to 150 ASCII characters (one line) are stored in consecutive locations for output.

```
ASSIGN 99 TO IFORM
CALL ENCODE (IBUF, IFORM, 3, LIST)
```

```
99  FORMAT(I3)
```

Where: LIST(1) = \$0023, LIST(2) = \$FFFE, LIST(3) = \$001A

Then: IBUF(1) = \$2033, IBUF(2) = \$3520, IBUF(3) = \$2D31,
IBUF(4) = \$2032, IBUF(5) = \$3600

If IBUF is output on the teletypewriter, the following results:

```
^35^-1^26
```

NOTE

In the preceding example the ASSIGN statement is used as the only way to set a variable (IFORM) equal to a statement number. The request also specifies that three variables are to be converted as three ASCII characters for a total of nine ASCII characters. Since each computer word contains two characters, IBUF must be dimensioned as a five-word data block. When the number of ASCII characters is odd, the last character on the teletypewriter results in no output.

```
DIMENSION LIST(7), FLIST(3), DATA(3,3), IBUF(50)
EQUIVALENCE (FLIST, LIST(2))
```

```
C 9  FLOATING POINT VARIABLES WERE COMPUTED AND SAVED IN DATA
```

```
ASSIGN 99 TO IFORM
99  FORMAT(5H LINE, I2, 3H X =, F5.2, 3H Y =, F5.2, 3H Z =, F5.2/)
    K=1
    DO 30 I=1,3
      LIST(1)=I
      DO 20 J=1,3
20   FLIST(J) = DATA(I, J)
      CALL ENCODE (IBUF(K), IFORM, 4, LIST)
30   K=K+16
```

The preceding example illustrates the mixing of floating-point with integer variables for ENCODE/DECODE calls. By equivalencing the floating-point variable name to the second entry of the integer

array, mixed values can be entered into the table. IBUF has been packed one line at a time (less than 150 characters per call) with a line feed indicated as the last character; however, the total number of characters packed in IBUF is 96. The following results when output:

```
LINE^1^X=^ 1.00^Y=^1.00^Z=^1.00
LINE^2^X=^ 0.52^Y=^3.42^Z=-1.50
LINE^3^X= 24.50^Y=-0.25^Z=50.20
```

9.4.2 ENCODE MACRO

The ENCODE subroutine may be called in assembly language by calling the ENCODE macro as follows:

```
ENCODE A, B, C, D, E,      (absolute)
      or
ENCODE* A, B, C, D, E     (relative)
```

Parameters A, B, C, D correspond directly with the respective parameters in a FORTRAN call, as shown above. Parameter E is the address of an error routine to which control is given. If E is blank no test for error conditions is made. (See Section 9.4.5.)

9.4.3 DECODE

DECODE transmits n consecutive ASCII characters according to the iform from locations starting with the first word in buffer to the variable list as n machine-language elements.

```
ASSIGN 99 TO IFORM
CALL DECODE (IBUF, IFORM, 10, LIST)
99  FORMAT(10I3)
```

In the preceding example, the ASSIGN statement is used as the only way to set a variable (IFORM) equal to a statement number. Also, the request specifies that ten integer values be stored in LIST as hexadecimal numbers. IBUF must contain 15 words of ASCII characters since a total of 30(10*13) characters with two characters per word were requested.

```
ASSIGN 99 TO IFORM
CALL DECODE (IBUF, IFORM, 5, LIST)
99  FORMAT (3(2X, 2I5))
```

Where: Five integer values are stored in LIST as hexadecimal integers even though the FORMAT specifies six integer values; that is, skip two characters, pick up the next five characters twice, and repeat this format twice.

```
ASSIGN 99 TO IFORM
CALL DECODE (IBUF, IFORM, 20, LIST)
99  FORMAT(3(3I2, 1X, 2I3))
```

NOTE

Even though 15 integers were specified in the FORMAT statement, a repeat of the FORMAT starting with the first specification within the parenthesized expression is executed to complete the conversion of the LIST parameters.

```
ASSIGN 99 TO IFORM
CALL DECODE (IBUF, IFORM, 0, 0)
99  FORMAT(28H (message))
```

Where: 28 ASCII characters are transmitted from IBUF to IFORM.

NOTE

This is a way of editing FORMAT statements without recompiling.

```
DIMENSION FLIST(8), LIST(18)
EQUIVALENCE (FLIST, LIST(3))
```

```
ASSIGN 99 TO IFORM
CALL DECODE (IBUF, IFORM, 10, LIST)
99  FORMAT(2I2, 8F10.3)
```

Where: The first two variables are integer values and the remaining eight are floating-point.

NOTE

Ten variables were specified even though the floating-point variables occupy two words per variable.

9.4.4 DECODE MACRO

The DECODE subroutine may be called from an assembly language program by calling the DECODE macro as follows:

```
DECODE A, B, C, D, E (absolute)
      or
DECODE* A, B, C, D, E (relative)
```

Parameters A, B, C, and D correspond to the respective parameters in a FORTRAN call, as shown above. Parameter E is optional and may be left blank. If defined, E defines the address of an error routine to which control is given when errors are detected.

9.4.5 ENCODE/DECODE ERROR DETECTION

When calling the ENCODE/DECODE package as subroutines, the error flag returned in the A register is lost. However, when calling the ENCODE/DECODE package as function subroutines, the error flag (= -1) returned may be tested for FORMAT errors except Fw.d output. The A register equals +0 on correct formatting of results.

```
CALL ENCODE (IBUF, IFORM, N, LIST)
```

In this example the ENCODE call is a subroutine call and the error flag (= -1) returned could not be tested.

```
NFLAG = ENCODE (IBUF, IFORM, N, LIST)
IF (NFLAG.EQ. -1) GO TO 1000
or IF (ENCODE (IBUF, IFORM, N, LIST).EQ. -1) GO TO 1000
```

In this example, the ENCODE call is a function subroutine call and NFLAG is set to the value returned by ENCODE to be tested later in an IF statement, or the error flag returned in the A register can be directly tested in an IF statement. In the function subroutine call, ENCODE/DECODE should be declared in a type statement as INTEGER; otherwise, the compiler treats the results returning from ENCODE/DECODE as floating point.

CAUTION

When ENCODE or DECODE is used in an implied DO loop, termination will occur immediately upon format conversion errors. Subsequent conversions within the loop will not occur.

9.4.6 ADDITIONAL FORMATTING ROUTINES

Additional formatting routines have been added to enable the FORTRAN programmer to format one variable at a time to save execution time needed for interpretation of FORMAT. The features in this section are to be used with one word integer type variables whenever integer type variables are used.

HEXASC and HEXDEC

CALL name(variable,buffer)

Where: name is HEXASC — Converts a hexadecimal integer to ASCII characters.
 HEXDEC — Converts a hexadecimal integer to a decimal integer in ASCII characters.
variable is the location of the hexadecimal integer.
buffer is the location of a two-word buffer to contain the converted integer in hexadecimal form (HEXASC); or the location of a three-word buffer to contain the converted integer in decimal form (HEXDEC).

Example 1:

```
DIMENSION LIST (10),IBUF(30)
      .
      .
      .
      J=1
      DO 10 I=1, 10
      CALL HEXDEC(LIST(I),IBUF(J))
10    J=J+3
      .
      .
      .
```

This call is comparable to an Iw FORMAT specification for output as I=1,2,3,...,10; J=1,4,7,...,28, and the subroutine HEXDEC fills IBUF with integer values from LIST.

```
      J=1
      DO 10 I=1, 10
      CALL HEXASC(LIST(I),IBUF(J))
10    J=J+2
```

This call is comparable to the \$w FORMAT specification for output. The subroutine HEXASC fills IBUF with ASCII values from LIST.

Example 2:

If IVAL = 258,
then CALL HEXASC(IVAL,IBUF(1))
results in IBUF(1) = \$3031
 IBUF(2) = \$3032
and CALL HEXDEC(IVAL,IBUF(1))
results in IBUF(1) = \$2020
 IBUF(2) = \$2032
 IBUF(3) = \$3538

HEXASC and HEXDEC Macros

The above subroutines may be called from assembly language programs by making these macro calls:

HEXASC A, B (absolute)
or
HEXASC* A, B (relative)
HEXDEC A, B (absolute)
or
HEXDEC* A, B (relative)

Where: A is the address of the variable.
 B is the address of the buffer (two words HEXASC, three words HEXDEC).

ASCII and DECHEX

CALL name(buffer, variable)

Where: name is ASCII — Converts two words of ASCII characters in BUFFER to a hexadecimal integer.
 DECHEX — Converts three words of a decimal integer in ASCII characters in BUFFER to a hexadecimal integer.
buffer is the starting location containing the ASCII representation of the integer.
variable is the location of the converted integer.

Example 1:

DIMENSION IBUF(25), LIST(10)
.
.
.
J=1

```

      K=1
      DO 10 I=1, 5
      CALL ASCII(IBUF(J), LIST(K))
      J=J+2
      K=K+1
      CALL DECHEX(IBUF(J), LIST(K))
      J=J+3
10    K=K+1

```

This example assumes that words 1, 6, 11, 16, and 21 in IBUF are hexadecimal and words 3, 8, 13, 18, and 23 in IBUF are decimal. The calls are comparable to \$w and Iw FORMAT specification (input), respectively.

Example 2:

```

If   IBUF(1) = $3030
     IBUF(2) = $3033
     IBUF(3) = $3035
then CALL ASCII(IBUF (2), IVALUE)
results in IVALUE = $305 = 773
and CALL DECHEX(IBUF(1), IVALUE)
results in IVALUE = $0131

```

ASCII and DECHEX Macros

The above subroutines may be called in assembly language by calling the appropriate macro as follows:

```

ASCII A, B (absolute)
ASCII* A, B (relative)

DECHEX A, B (absolute)
DECHEX* A, B (relative)

```

Where: A is the buffer address (two words for ASCII and three words for DECHEX)
 B is the variable address

AFORM and RFORM

CALL name(buffer, variable)

Where: name is AFORM — Converts a word containing two ASCII characters to two words each containing a character left-justified blank-filled
 RFORM — Converts a word containing two ASCII characters to two words each containing a character right-justified zero-filled.
 buffer is the location containing two ASCII characters
 variable is two words containing the resultant of AFORM/RFORM

Example:

```
DIMENSION LIST(2),IBUF(10)
      .
      .
      .
DO 10 I=1, 10
CALL RFORM(IBUF(I), LIST)
IF(LIST(1).EQ$2E) GO TO 20
IF(LIST(2).EQ.$2E) GO TO 20
10  CONTINUE
20  .
      .
      .
```

In this example, LIST is being scanned for an ASCII period. AFORM/RFORM are comparable to Aw and Rw respectively, as used on an input record.

FLOATG

CALL FLOATG (variable, buffer)

Where: variable is a two-word floating-point variable
buffer is a six-word output buffer containing the floating-point representation with its exponent in ASCII characters: ±.XXXXXXE±ee (this is equivalent to Ew.d FORMAT specification with d=6 and 0 ≤ ee ≤ ±39)

FLOATG Macro

The subroutine may also be called in assembly language by using a macro call as follows:

```
FLOATG A, B      (absolute)
FLOATG* A, B     (relative)
```

Where: A is the address of a floating-point variable
B is the address of a buffer (six words)

9.5 FORTRAN/MONITOR RUN-TIME PACKAGE

The FORTRAN/monitor run-time package was written to give the FORTRAN programmer a means of communicating with the monitor. It is necessary for the programmer to make certain monitor requests, obtain monitor parameters, or execute I/O commands.

The monitor requests are:

READ
WRITE
FORMAT-READ
FORMAT-WRITE
SCHEDULER
TIMER
RELEASE Memory

The READ, WRITE, FORMAT-READ, and FORMAT-WRITE requests were provided as a supplement to the FORTRAN READ/WRITE statement processor (Q8QIO).

The monitor request for FORMAT-READ or FORMAT-WRITE has a different interpretation than the FORTRAN formatted records.

Consult the MSOS reference manual for device driver characteristics with READ, FREAD, WRITE, and FWRITE calls.

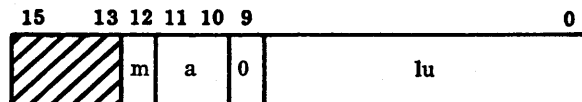
9.5.1 READ/WRITE CALLING SEQUENCE

CALL name (mlu, buffer, length, completion, flag, temp)

Where: name is READ, WRITE, FREAD, or FWRITE.

mlu is the mode and logical unit.

mlu format:



- m mode (used only on devices capable of both modes)
 - 0 Binary mode
 - 1 ASCII mode
- a logical unit designator
 - 0 actual logical unit number
 - 1 core location containing the logical unit number
- lu logical unit (as defined for the MSOS configuration)

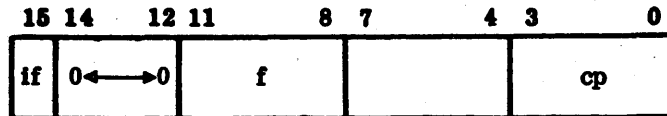
buffer is the area of memory that data is read into or written from

length is the number of words to be read or written. If this is a mass storage logical unit, then length is the name of a three-word table containing:

- LENGTH(1) Number of words
- LENGTH(2) Mass storage address (bits 15 through 30)
- LENGTH(3) Mass storage address (bits 0 through 14)

completion is the statement label or index to the system directory or external core-resident main program that is scheduled on completion of the request.

flag is the packed word with the completion priority in bits 0 through 3, the request priority in bits 4 through 7, and a flag in bits 8 through 11. The flag interpretation is:



if Indirect flag

f Flag. If set, the buffer address is contained in the location specified in the calling sequence.

- 0 completion = Statement label
- 1 completion = Index to the directory
- 2 completion = External core-resident main program

cp Completion priority (levels 0 through 15)

rp Request priority (levels 0 through 15)

temp is the eight-word area used for building the calling sequence to the monitor.

NOTE

Calls from the background for READ or WRITE mass storage requests must ensure that LENGTH(3) does not access scratch sector 0. LENGTH(3) must be ≥ 96 words.

MODE AND LOGICAL UNIT

The logical unit of the device or a core location containing the logical unit number is in bits 9 through 0 of lu. If bit 11 equals 0, then bits 9 through 0 are the actual logical unit. If bit 11 equals 1, then bits 9 through 0 are a core location containing the logical unit. The mode indication is bit 12 (= 1 ASCII; = 0 unformatted or binary). The core locations containing the standard input/output logical units as defined in the monitor are detailed as follows:

<u>Core Location₁₆</u>	<u>lu Format₁₆</u>	<u>Meaning</u>	<u>Mode</u>
F9	18F9	Input medium	ASCII
	08F9	Input medium	Binary
FA	18FA	Output punch medium	ASCII
	08FA	Output punch medium	Binary
FB	18FB	Output list medium	ASCII
FC	18FC	Output comment	ASCII
FD	18FD	Input comment	ASCII
C2	08C2	Mass storage	Binary†

COMPLETION LOCATION AND FLAG PRIORITY

When I/O has finished, control is returned to the completion location assigned at the time of the request.

The completion location may be a statement label in the same program (flag = 0). In FORTRAN, the only way to set a statement label as a completion location is with the ASSIGN statement.

Example: Completion = statement label

```

ASSIGN 100 TO ICOMPL
IFLAG = 0
CALL FWRITE (LU, IBUF, LENGTH, ICOMPL, IFLAG, ITEMP)
.
.
.
100 CONTINUE

```

The completion location may be a program residing in the system library (flag = 1). An EXTERNAL statement is used to correctly define the name of the program in the system library, and the loader inserts the index to the system directory as the completion location.

Example: Completion = system program

```

EXTERNAL NAME1
IFLAG=$100
.
.
.
CALL FWRITE (LU, IBUF, LENGTH, NAME1, IFLAG, ITEMP)

```

†When writing ASCII information on mass storage the mode is ignored.

The completion location may be the name of another program in core (flag = 2). An EXTERNAL statement is used to define the name of the program correctly. The completion location may never be a subroutine.

Example: Completion = external core-resident program

```
EXTERNAL NAME1
IFLAG=$200
.
.
CALL FWRITE(LU,IBUF,LENGTH,NAME1,IFLAG,ITEMP)
```

Example: Indirect buffer address

```
ASSIGN 10 TO IBUF
IBUF=IBUF+2
IFLAG=$8012
CALL FWRITE(LU,IBUF,LENGTH,ICOMPL,IFLAG,ITEMP)
10  FORMAT(50H THIS IS AN EXAMPLE OF AN INDIRECT BUFFER ADDRESS )
```

NOTE

In the preceding example the address IBUF was updated two words to remove "50H" from the message. Also an even number of characters does not include the terminating).

Alternatively, the FORMAT statement may be written as

```
10  FORMAT (^THIS IS AN EXAMPLE OF AN INDIRECT BUFFER ADDRESS^ )
```

NOTE

The alternative form also requires that the address IBUF be updated by two words to skip over the compiler generated "50H" in the buffer.

Example: (direct buffer address)

```
DIMENSION IBUF(4), ITEMP(8)
DATA IBUF / ^MESSAGE' /, LENGTH /4/, LU /$18FB/
IFLAG = $12
CALL FWRITE (LU, IBUF, LENGTH, ICOMPL, IFLAG, ITEMP)
```

9.5.2 SCHEDULER AND TIMER

REQUESTS

In a given system, numerous requests for the execution of programs at specific priority levels may be generated. Specifically these requests are generated when:

- An I/O request has been completed
- A specified time interval has elapsed
- Core has been allocated/released
- A mass storage request has been executed

Requests may also be made directly by making a scheduler call. It is the function of the scheduler request processor to:

- Cause the immediate execution of a program if it is of a higher priority level than the current program
- Thread the request by priority and on a first-in/first-out (FIFO) basis if its priority is equal to or lower than the current priority.

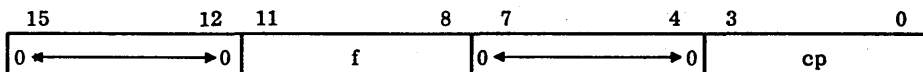
If the requested program is mass storage resident, the scheduler request processor causes allocation of core for this program and transfer of the program from mass storage. After the program has been transferred, a scheduler request is made, which results in one of the above.

Whenever a program terminates, the dispatcher selects the next program to be run, either from the top of the scheduler thread or the interrupt stack.

CALLING SEQUENCE

CALL SCHEDL(l, flag, parameter, temp)

Where: l is the requested program to be scheduled at the completion priority.
flag is a packed word with the completion priority in bits 3 through 0 and a flag in bits 11 through 8. The flag interpretation is:



f is the flag

0 L is a statement label

1 L is an index to the directory

2 L is an external core-resident main program

cp is the completion priority (levels 0 through 15)

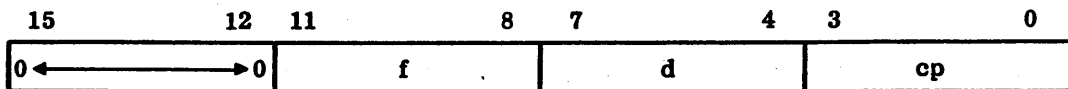
parameter is a positive integer may be passed to the scheduled program. The scheduled program obtains the parameter by calling the integer function LINK.

temp is a four-word area in which the scheduler call is generated. After the scheduler call is complete, this area is available for other use.

CALL TIMER(l, flag, time, temp)

Where: l is the program given control at priority CP after the time interval has expired.

flag is a packed word containing the completion priority in bits 3 through 0, a unit of time code in bits 7 through 4, and a flag in bits 11 through 8.



f is the flag

0 L is a statement label

1 L is an index to the directory

2 L is an external core-resident main program

d is the unit of time

0 Counts of system time base

1 0.1 second

2 1.0 second

3 1.0 minute

cp is the completion priority

time is the time interval to delay before scheduling the program, l, at level CP. At the end of the time interval, the core clock (contents of cell \$E8) is passed to the requested program as a parameter. To obtain this parameter the integer function LINK must be called.

temp is a four-word area in which the timer call is generated. After the call has been executed, this area is free for other use.

9.5.3 MISCELLANEOUS CALLING SEQUENCES

LINK

N = LINK (0)

N is set to:

- The passed parameter from a scheduler call if LINK is called at the start of the scheduled program.
- The value of the core clock if LINK is called at the start of the program called by a TIMER request.
- The error flag at the completion of I/O if LINK is called at the completion location.

DISPATCHER

CALL DISPAT or CALL DISP

Control is given to the dispatcher in the monitor to start the next highest priority program.

CORE CLOCK

The integer function ICLOCK obtains the value of the clock: I=ICLOCK(0)

I contains the current value of the clock (memory location \$E8).

RELEASE OF ALLOCATED CORE

All programs that have been allocated core (either allocatable or partition) must return memory to the core allocator when they are finished. This includes all mass-storage-resident programs.

CALL RELESE (main)

Main is the name of the main program. The call must be compiled as the last executed statement in the program. No further program statements are executed following CALL RELESE.

OUTPUT COMMANDS VIA THE A/Q CHANNEL

CALL OUTINS (IOUTAQ)

Where: IOUTAQ is a three-word table

IOUTAQ(1) is loaded into the Q register. Should contain converter, equipment, and station codes or the channel addresses for a connect command.

IOUTAQ(2) is loaded into the A register. Contents vary depending upon the device selected.

IOUTAQ(3) is a flag word which contains the following information after the call

- 0 No reject
- 1 Internal reject
- 2 External reject

INPUT COMMANDS VIA THE A/Q CHANNEL

CALL INPINS(IINAQ)

Where: IINAQ is a three-word table

IINAQ(1) is loaded into the Q register. Should contain converter, equipment, and station codes or the channel addresses for a connect command.

IINAQ(2) is after the call, contains the data or status obtained on input.

IINAQ(3) is a flag word which contains the following information after the call

- 0 No reject
- 1 Internal reject
- 2 External reject

CONNECT THE 1750 DATA AND CONTROL TERMINAL AND INPUT

CALL ICONCT(IINAQ)

Refer to input commands via the A/Q channel for the calling sequence interpretation.

CONNECT THE 1750 DATA AND CONTROL TERMINAL AND OUTPUT

CALL OCONCT(IOUSAQ)

Refer to output commands via the A/Q channel for the calling sequence interpretation.

9.5.4 BUFFERED INPUT/OUTPUT

There are many ways of accomplishing asynchronous I/O operation (for example, reading/writing from one buffer while executing from another) with programming techniques. The following example demonstrates how this may be accomplished.

```
DIMENSION DATA (100, 2), TEMP(4)
      :
      :
      ASSIGN 100 TO ISTART
1.    IPRIOR=4
      CALL SCHEDL(ISTART, IPRIOR, 0, TEMP)
      CALL DISPAT
      100 KX=1
      ASSIGN 200 to ICOMPL
2.    CALL READ(LU, DATA(1, KX), 100, ICOMPL, IPRIOR, TEMP)
```

```

3.    CALL DISPATCH
4.  200 IF(LINK(0). LT. 0) GO TO 300
      JX=KX
      KX=3-KX
5.    CALL READ(LU, DATA(1, KX), 100, ICOMPL, IPRIOR, TEMP)
      .
      .
6.    .      COMPUTE WITH DATA(1, JX)
7.    CALL DISPATCH
      .
      .
C    ERROR HANDLING SECTION
8.  300 CONTINUE
      .
      .
      END

```

Following is an explanation of the preceding coding.

1. Make a scheduler call to set the priority level of the program.
2. Initialize a READ of n words (≤ 100 words) into DATA(1, KX) where KX=1.

NOTE

The first word address is DATA(1) and the priority level for this READ is the same as the program.

3. Make a dispatcher call. Another program is given control until the I/O is complete.
4. The completion location (200) tests for errors in reading the input data, then switches JX and KX.
5. Initialize a second READ of n words (≤ 100 words) into DATA (1, KX) where KX=2.

NOTE

FWA is DATA(101) and the completion priority must be at the same level as the program priority level.

6. Execute the code using the data in the filled buffer where JX indicates which buffer is filled.
7. When execution of DATA(1, JX) is complete, call the dispatcher.
8. If the filling buffer (KX) is complete, control goes to the completion address where the buffers are switched and the sequence of operations is restarted. However, if the filling buffer (KX) is incomplete, control remains with another lower priority program, while I/O is in progress, before returning to the completion address.

This section describes the macro facility, which is a macro card-deck feature. This facility allows the user to read in a deck of cards, save them on mass memory, and then insert them later in any program. After a macro has been defined, it remains in the macro library for the length of the stacked compilation.

This feature is very useful when working on large program systems. If the system includes many routines using the same common block with the same format, etc., those cards may be defined as a macro and then inserted in each program by using only one card. Any changes that need to be made may then be made only to the macro definition deck.

10.1 MACRO DEFINITION

A macro deck is an entity similar to a program, subroutine, function, or block data. It begins with the statement:

MACRO name

and ends with the statement:

END

Example: MACRO MASTER
 C THIS IS THE MASTER LABELED COMMON BLOCK
 COMMON /A/ IFLAGS, LINECT, LINCT1
 COMMON /A/ . . . ETC.
 .
 .
 .
 ETC.
 .
 .
 .
 END

10.2 MACRO CALL

To use a macro, an M is placed in column 1 of the card, followed by the name of the macro anywhere on the card.

Example: SUBROUTINE PHASE A
 C THIS IS THE PHASE A PROCESSOR
 M MASTER
 M UNLABL
 .
 .
 .
 ETC.
 .
 .
 .

10.3 RESTRICTIONS

The END statement is used to end the macro definition but is not part of the macro itself.

A macro may not include a call to another macro.

The macro must be defined before being referenced by a program, subroutine, or function. The macro does not have to be the first program in a compile.

10.4 LIMITS

Maximum number of macros: 32
Maximum number of cards in one macro: 255
Maximum number of cards for all macros: 319

GLOSSARY

Actual argument	An argument in the list of a calling program that represents actual values relative to the calling program.
Arithmetic expression	A combination of arithmetic operators and data elements that produces a single numerical value when evaluated by execution.
Assignment statement	An executable statement that gives a numerical value to a variable. The value may be the result of calculation or it may be the result of calculation or it may be assigned by the programmer.
Basic external function	An auxiliary procedure that supplies references to library routines contained in the system. The basic external function is used to evaluate more complex mathematical evaluations.
Block data subprogram	A subprogram without reference that supplies specifications and initial values to labeled common.
Byte	A sequence of adjacent binary digits operated up on as a unit and usually shorter than a computer word.
Common	An area of memory that may be shared between program units.
Compiler	A program that translates a programming language (such as FORTRAN) into an assembly language and, often, into machine language. A compiler may generate many machine instructions for a single symbolic statement.
Constant	A value assumed to be fixed or invariable in a given operation or calculation.
Control statement	An executable statement used to alter the normal sequence of program execution (i.e., from one statement to the following statement) or to cause a number of iterations of a program section.
Data element	A single-valued unit of data which may be uniquely referenced. It may occupy part of a word, a full word, two full words or three full words.
DATA statement	A nonexecutable statement used to assign constant values to variables or arrays at the time of compilation.
Data type	The six types of data that are recognized by MS FORTRAN; i.e., integer, single, real, double precision, byte, and signed byte data types.

Double precision	The use of three computer words to represent a number.
Dummy argument	An argument listed in the procedure definition that serves to exchange values between the reference list and the procedure calculations.
Executable statement	A statement that performs calculations, directs control of the program, and transfers data.
Execution	The process in which the instructions contained in a program direct the activities of the control processing unit.
External function subprogram	A separate FORTRAN program which performs a set of calculations when its name appears in an arithmetic or logical expression in the referencing program. The external function subprogram is used when a number of calculations are required to obtain a single result.
External subroutine subprogram	A separate FORTRAN program that performs a calculation in conjunction with another program it calls into operation. The external subroutine subprogram is used when a number of calculations are required to obtain an array of values.
Field	In a record, a specified area used for a particular category of data.
Field descriptor	A descriptor used in FORMAT statements to define the field of each element in the I/O list in formatted READ and WRITE statements.
Field separator	A slash or comma used to separate field descriptors or groups of field descriptors.
File manager	A general-purpose file management package that consists of a request supervisor and a number of request processors.
FORMAT statement	A nonexecutable statement that defines the field and data type of each element in the I/O list.
FORTRAN/monitor run-time package	The package that gives the programmer a means of communicating with the monitor.
FORTRAN READ/WRITE statement processor	The processor that transfers information between core storage and external peripheral devices connected to the computer.
Function	A procedure that supplies a single result to be used at the point of reference.
In-line code method	A method used if an intrinsic function involves only a few machine instructions. The intrinsic function is inserted in the referencing program every time the reference appears.

Intrinsic function	An auxiliary procedure that supplies predefined calculations by inserting simple sets of calculations into the object program at compile time. The intrinsic function is used to evaluate frequently used mathematical functions.
I/O statement	An executable statement that does one of the following: reads records from an external unit into core and writes records out of core onto an external unit, or affects the position and checks the status of external magnetic tape files.
Logical expression	A combination of relational expressions and logical operators such that evaluation of the expression produces a result of true or false.
Logical unit	A number that can be equated to any one of a variety of peripheral units.
Mass storage file	A file that is assigned to the scratch area of the mass storage device and is not retained after execution of a job.
Mixed mode expression	An arithmetic expression which can include integer, real, and double precision quantities.
Multiprogramming	In MSOS, a technique for processing two programs simultaneously by overlapping or interleaving their execution. Multiprogramming is achieved by allowing the priority program to gain control of the processor periodically through interrupts.
Nonexecutable statement	A statement that provides the compiler with information regarding data structure and storage.
Nonre-entrant ENCODE/DECODE run-time package	A run-time library that runs in the background and has multiprogramming characteristics. It is designed for use in debugging programs.
Re-entrant ENCODE/DECODE run-time package	A run-time library that runs in the foreground and has multiprogramming characteristics.
Re-entrant program	A program that can be interrupted and re-entered by another program of higher priority level.
Relational expressions	Two arithmetic expressions combined with a relation operator such as .EQ (equal to), .NE (not equal to), etc.
Run-anywhere program	A program that can be moved and successfully operated elsewhere in core after being loaded by a relocating loader.
Specification statement	A nonexecutable statement that specifies type, word structure, and storage for variables.
Statement function	An auxiliary procedure that is defined by a single statement in the program unit in which it is referenced. The statement function is used when a single nonstandard computation is needed repeatedly.

Standard logical limits	Standard logical units for input, output and listing are stored in system memory. Fortran calls these by referencing units 1, 2, and 3 respectively. Logical unit 4 is the comment device.
Subprogram	A user-defined set of statements compiled independently of the program unit that references it or to which it supplies specifications and initial values.
Supplied function	A predefined calculation or reference to library routines in the system that are supplied by MS FORTRAN to evaluate standard mathematical functions.
Symbolic name	A data name or procedure name that consists of one to six alpha-numeric characters, the first of which must be alphabetic.
Variable	A quantity that can assume any of a given set of values.

COMMUNICATION BETWEEN FORTRAN AND ASSEMBLY LANGUAGE PROGRAMS

A

THE FORM OF THE CALLING SEQUENCE

Calling sequences written in assembly language which are intended to communicate with FORTRAN-generated subprograms must have the following form, where SUB has been previously declared as an external.

LOC	RTJ	SUB
LOC+1	(RTJ SUB is a two-word instruction)	
LOC+2	Address of argument 1	
LOC+3	Address of argument 2	
LOC+4	Address of argument 3	
⋮		
LOC+N+ 1	Address of argument N	
LOC+N+ 2	Program resumes	

When a function subprogram returns a floating-point value, the result is placed in the pseudo accumulator and may be accessed by means of a call to the floating-point package.

The result of an integer function is left in the A register.

Addresses of arguments occur in consecutive locations following the RTJ command, one cell per address, in the order that the arguments appear in the actual parameter list which should be the same subprogram definition. Subroutines need not necessarily have arguments.

FORTTRAN calls to assembly language subroutines must recognize the argument passing sequence as previously described. The arguments must have the same order as their use and are assembled in the form as previously shown. When a call to a routine outside of a FORTRAN program is made and I/O is performed, a priority problem may be encountered. In such a case the priority of FORTRAN I/O and other devices used should be examined to determine if a higher priority device has interfered.

ABSOLUTE ADDRESSES

All arguments in common are in the calling sequence as absolute addresses. In a non-run-anywhere program, all arguments are absolute.

RELATIVE ADDRESSES

Relative addressing is only used in programs compiled under the R option.

All arguments which do not fall into the category for absolute addresses are represented in the calling sequence as relative addresses. The self-relative address (which is what is meant by a relative address in a calling sequence) is computed by subtracting the location of the self-relative address in the calling sequence, say LOC+3, from the address of the corresponding argument, say argument 3, and setting bit 15 to 1.

Only the 15 low order bits of an argument (14 through 0) are necessary to designate the address absolute or relative. Thus, in calling sequences, bit 15 is used as a flag to distinguish between the two addressing modes.

Bit 15 is 0 if argument address is absolute.

Bit 15 is 1 if argument address is relative.

The address returned from a floating-point calling sequence is absolute.

FORTTRAN assumes that all assembly language routines save and restore the Q and I registers.

TABLE CAPACITIES

B

FORTRAN TABLE LIMITS

Up to 2,340 compiler-generated and user symbols are allowed.

Up to 10 nested DO loops are allowed.

The maximum number of declared subscripts is 147.

The maximum number of continuation cards allowed per statement is five.

No more than 30 parenthesis levels are allowed.

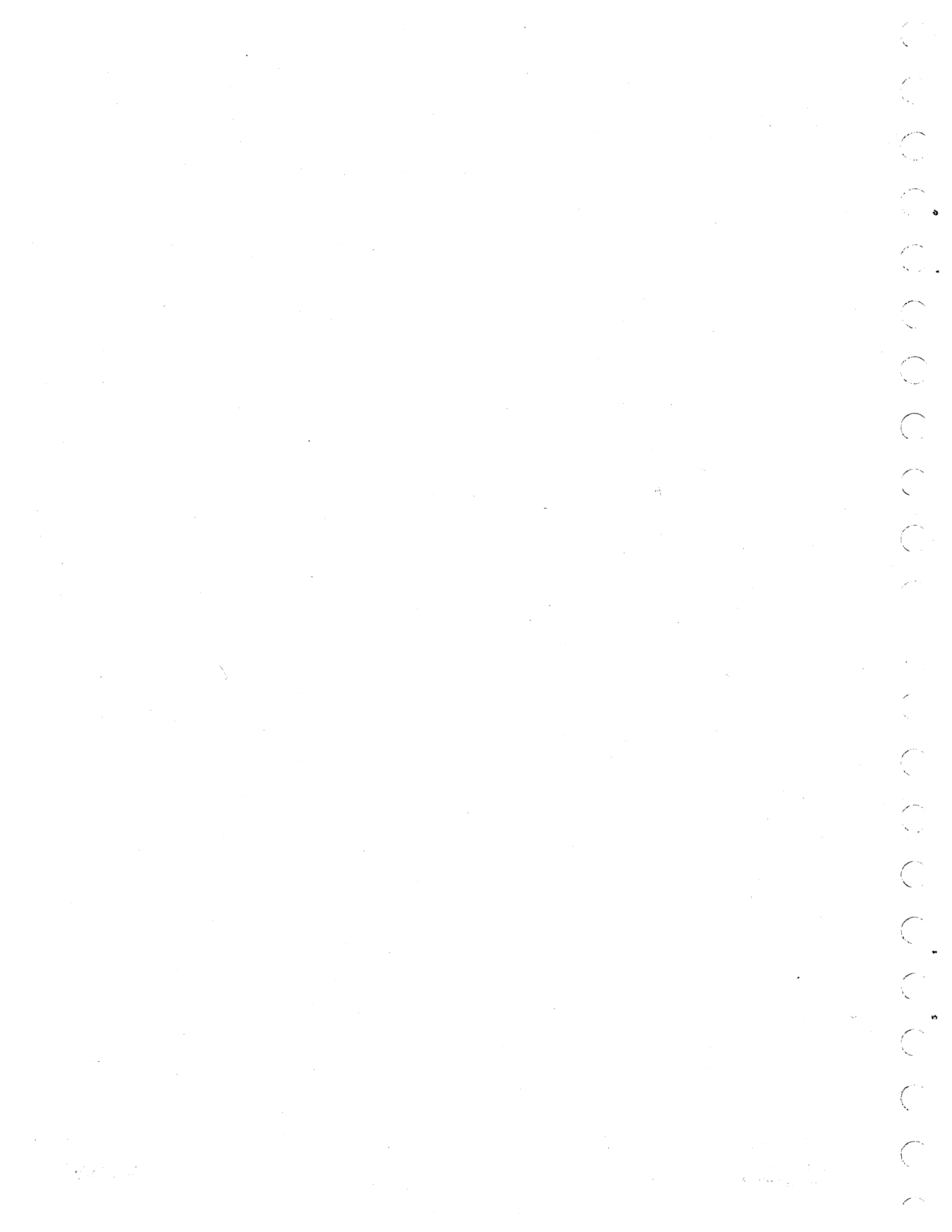
The number of unique dummy argument index constant pairs must not exceed 50.

The number of subroutine arguments may not exceed 50.

Literals in DATA statements are limited to 387 characters.

Up to 51 EQUIVALENCE relations are allowed.

The number of compiler-generated words may not exceed 300 per source statement, or else a compiler table overflow error F, 100 will be generated.



SINGLE-PRECISION FLOATING-POINT PACKAGE

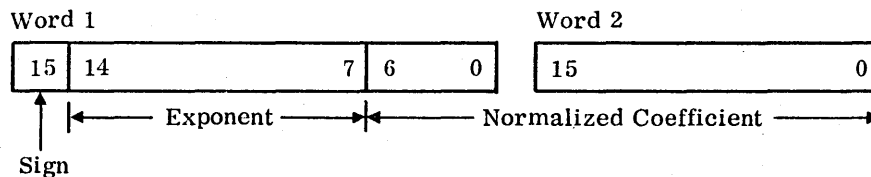
C

The software single-precision floating-point package used by FORTRAN is described in this appendix. The package also can be used by an assembly-produced program.

Two similar floating-point packages are called by the same name (HFLOT); one is re-entrant and the other is not. Both packages are usable by run-anywhere programs. The re-entrant package must operate in protected core; the other package may operate anywhere.

If the firmware version of the package is used, the call to HFLOT is vectored to the firmware where equivalent functions are performed.

Each floating-point number requires two consecutive words of storage. The first word (most significant bits) is the one that is addressed. Normalized floating-point format is as follows:



A floating-point number x is in the range given in the following example and is significant to one part in eight million.

$$-2^{127} (1-2^{-23}) \leq x \leq 2^{127} (1-2^{-23})$$

If the most significant word is zero (16 bits of zero or one), a floating-point zero is assumed.

COEFFICIENT

The coefficient consists of a 23-bit number n , $1-2^{-23} > |n| > 0$. The high-order bit position of the first word is the coefficient sign bit. A zero denotes a positive coefficient and a one denotes a negative coefficient. When the coefficient is negative, the entire floating-point number, exclusive of the sign bit, is stored in complement form.

EXPONENT

The floating-point exponent is an eight-bit quantity ranging from 00 to FF₁₆. Through biasing by 80₁₆, this range expresses both positive and negative exponents. The biasing is accomplished according to the following rules:

1. If the floating-point number is negative, complement the entire floating-point word and remember that the number is negative. The exponent is now in a true biased form.

2. If the biased exponent is equal to or greater than 80_{16} , subtract 80_{16} to obtain the true exponent. If less than 80_{16} , subtract $7F_{16}$ to obtain the true exponent. (Observe the algebraic rules for subtraction.)
3. Separate the coefficient and exponent. If the true exponent is negative, move the binary point left the number of bit positions indicated by the true exponent. If the exponent is positive move the binary point right the required number of places.
4. The coefficient has now been converted to fixed binary. The sign of the coefficient is negative if the original floating-point number was complemented in step 1. The sign bit must be extended if the quantity is to be placed in a register.
5. Convert the quantity to decimal representation by using the Powers method.

Example 1:

Floating point number BF BF FFFF
 IN BINARY 1011 1111 1011 1111 (FFFF)
 NEG ∙• COMPLEMENT: 0100 0000 0100 0000 (0000)
 EXTRACT EXPONENT: 100 0000 0

CONVERT TO HEX: 80_{16}

$$\begin{array}{r} \text{UNBIAS: } 80 \\ \underline{-80} \\ 0_{16} \end{array}$$

NORMALIZED COEFFICIENT EQUALS

$$.100\ 0000_2\ (0000)$$

NO BINARY POINT MOVEMENT NEEDED USING POWERS RULES

$$.1x2^{-1} + 0x2^{-2} + 0x2^{-3} + \dots + Ax2^{-n}$$

DOING ARITHMETIC

$$.1/2 + 0 + 0 + \dots + 0$$

$$\text{ANSWER} = -.5_{10}$$

Example 2:

Floating point number 3BC0 0000
BINARY = 0011 1101 1100 0000 (0000)
 011 1101 1

$$\begin{array}{r} \text{EXPONENT} = 7\text{B}_{16} \\ \quad \quad \quad 7\text{B}_{16} \\ \quad \quad \quad \hline \quad \quad \quad -7\text{F} \\ \quad \quad \quad \hline \quad \quad \quad -4 \end{array}$$

MOVE BINARY POINT LEFT 4 PLACES

$$\begin{array}{l} .100\ 0000\ (0000) \\ .0000100\ 0000 \\ = .0x2^{-1} + 0x2^{-2} + 0x2^{-3} + 0x2^{-4} + 1x2^{-5} + \dots \\ = .2^{-5} \end{array}$$

$$\text{ANSWER} = \frac{1}{32} = .03125_{10}$$

Example 3:

Floating point number 44C0 0000
BINARY 0100 0100 1100 0000 (0000)
Exponent 100 0100 1

$$\begin{array}{r} 8\ 9_{16} \\ 89 \\ \hline -80 \\ \hline +9 \end{array}$$

MOVE BINARY POINT RIGHT 9 PLACES

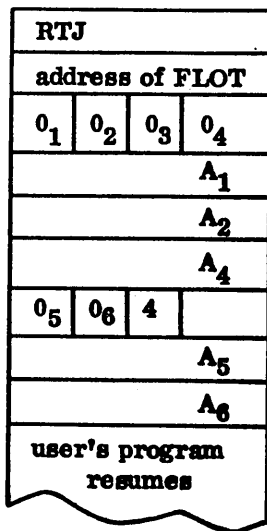
$$\begin{array}{l} .100\ 0000\ (0000) \\ 100\ 0000\ 00. \\ 1x2^8 + 0x2^7 + \dots + 0x2^0 \quad .0x2^{-1} + \dots + Ax2^{-n} \end{array}$$

$$\text{ANSWER} = 2^8 = 256_{10}$$

CALLING SEQUENCE

FLOT uses an interpretive calling sequence. Neither calling sequence saves Q or I, nor uses the communication cells. In the re-entrant cases, the communication cells must be saved upon entrance to a program unit and restored upon exit (it is the user's responsibility to save these communication cells). The interpretation is on a string of four-bit bytes, where the leftmost four-bit byte represents the first operation. The respective operands, if they exist, are in the same order as the bytes, with one operand per byte. As many operations as desired may exist, but the last one must be the terminator of a four. The floating-point accumulator is retained between calls to FLOT.

Example:



The calling sequence was designed to minimize core requirements, including core used to set up the calling sequence.

OPERATIONS

The following operations are used by the floating-point package.

<u>Operation</u>	<u>4-Bit Code</u>	<u>Meaning</u>
SPEC	0	Special. This is first byte of a two-byte operation code.
FLOF	1	Float to fix. The pseudo accumulator is converted to a 16-bit integer and stored at the effective operand address.
FIXF	2	Fix to float. The contents of the effective operand address are converted from a 16-bit integer number and placed in the pseudo accumulator.

<u>Operation</u>	<u>4-Bit Code</u>	<u>Meaning</u>
STRI	3	Store index. The index register is stored at the effective operand address.
FEND	4	End of calling sequence. This operation terminates the calling sequence. No operand needed.
CHMD	5	Change mode of operation. All operand addresses following this operation code in the calling sequence are made relative if the preceding addresses were absolute, or absolute if the preceding addresses were relative. Addresses are initially absolute. No operand is needed.
NIDX	6	No index. The succeeding operands do not have indexing increments. NIDX supersedes any preceding INDX and is superseded by any following INDX. NIDX is assumed initially. No operand is needed.
FCOM	7	Floating complement. The pseudo accumulator is complemented. No operand is needed.
FSUB	8	Floating subtract. The contents of the effective operand address are subtracted from the pseudo accumulator and the result is left in the pseudo accumulator.
FMPY	9	Floating multiply. The pseudo accumulator is multiplied by the contents of the effective operand address and the result is left in the pseudo accumulator.
FDIV	A ₁₆	Floating divide. The pseudo accumulator is divided by the contents of the effective operand address and the result is left in the pseudo accumulator.
FLDD	B ₁₆	Floating load. The floating-point number in the corresponding effective operand address is transferred to the pseudo accumulator.
ADDI	C ₁₆	Add to index. The contents of the effective operand address are added to the index register.
FLST	D ₁₆	Floating store. The floating-point number is transferred from the pseudo accumulator to the corresponding effective operand address.
FADD	E ₁₆	Floating add. The contents of the effective operand address are added to the pseudo accumulator, and the sum is left in the pseudo accumulator.
INDX	F ₁₆	Index. The operand corresponding to INDX is used to increment the operand of the following operations: FLDD, FLST, FADD, FSUB, FMPY, and FDIV. Each succeeding INDX supersedes the last. No index is initially assumed.

The following operation codes are executed only if the preceding byte is a SPEC(0).

<u>Operation</u>	<u>4-Bit Code</u>	<u>Meaning</u>
CACS	1	Continue another calling sequence. The next operation to be executed is the most significant four-bit command code located at the effective operand address (unconditional branch).
BRAM	2	Branch accumulator minus. If the pseudo accumulator is minus, control is transferred to the effective operand address.
BRAZ	3	Branch accumulator zero
BRAN	4	Branch accumulator nonzero
BRAP	5	Branch accumulator positive
BRIM	6	Branch index register minus
BRIZ	7	Branch index register zero
BRIN	8	Branch index register nonzero
BRIP	9	Branch index register positive

NOTE

Codes 0 and A_{16} minus F_{16} , when preceded by a SPEC code, are executed as FEND.

RELATIVE ADDRESSING

The operand address, bits 15 through 0, is relative to self. The relative address is computed by subtracting the calling sequence operand address from the actual operand address.

Example:

$$X = -((A(I) + B(I) * C(I)) + (D(J) * E (J)))$$

Assume TEMP, X, J, D, and E are absolutely addressed and the other operands relatively addressed. The call to FLOT would look like the following.

RTJ		FLOT	
F ₁₆	B ₁₆	9	6
absolute address J			
absolute address D			
absolute address E			
D ₁₆	5	F ₁₆	B ₁₆
absolute address TEMP			
relative address I			
relative address A			
E ₁₆	9	7	6
relative address B			
relative address C			
5	E ₁₆	D ₁₆	4
absolute address TEMP			
absolute address X			

FAULT CONDITIONS

At any time during execution, the following fault conditions are flagged: exponent overflow, exponent underflow, and divide fault (attempted division by zero). These fault conditions may be tested and reset by use of the IFALT function call.

FLOATING-POINT ARITHMETIC WITH 23-BIT NUMBERS

A classic and straightforward technique is presented that is not limited to the size or type of the number representation used.

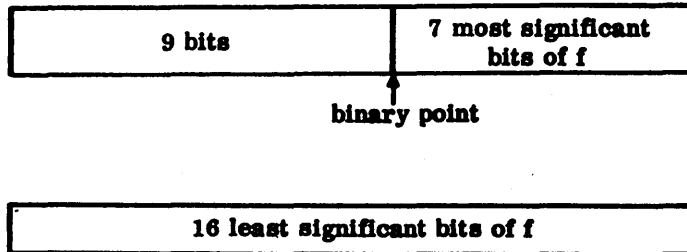
Consider the double-precision floating point number:

$$F = f \times \beta \tag{1}$$

where $|f|$ lies in the range

$$1/2 \leq |f| \leq 1 - 2^{-23} \tag{2}$$

Assume that we have a machine with a word length of 16 bits and that the 32 bits in the double-length word are divided in the following standard way:



The leftmost block of nine bits is divided into three parts:

- The first (leftmost) bit represents the sign of f .
- The second bit represents the sign of β .
- The next seven bits represent the magnitude of β .

This allows 23 bits for the representation of f . Assume that the binary point lies at the left of the 23 bits representing f so that the seven most significant bits of f are stored in the first word of the pair and the 16 least significant bits of f are stored in the second word of the pair.

H

$$|f| = c + d \times 2^{-7} \tag{3}$$

where c lies in the range

$$1/2 \leq c \leq 1 - 2^{-7} \tag{4}$$

and where d lies in the range

$$1/2 \leq d \leq 1 - 2^{-16} \tag{5}$$

then c represents the seven most significant bits of f and d represents the 16 least significant bits of f .

FOUR ARITHMETIC OPERATIONS

We wish to consider the four basic arithmetic operations using double-precision floating-point numbers of the form discussed. Consequently, in order to have notation for two operands, consider a second double-precision floating-point number

$$G = g \times 2^\delta \tag{6}$$

where $|g|$ lies in the range

$$1/2 \leq |g| \leq 1 - 2^{-23} \tag{7}$$

If

$$|g| = a + b \times 2^{-7} \tag{8}$$

where a lies in the range

$$1/2 \leq a \leq 1 - 2^{-7} \tag{9}$$

where b lies in the range

$$1/2 \leq b \leq 1 - 2^{-16} \tag{10}$$

then a represents the seven most significant bits of $|g|$ and b represents the 16 least significant bits of $|g|$.

Assume that the machine represents negative numbers using a one's complement system. Assume that the procedure for changing the sign of a double-precision floating-point number is to perform a bit-by-bit complement of the entire 32 bits (including the nine bits representing the sign and exponent).

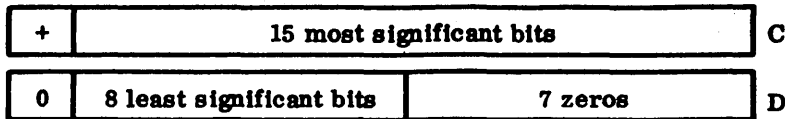
Multiplication

$$\begin{aligned} F \times G &= (f \times 2^\beta) (g \times 2^\delta) \\ &= (\text{sign } F \times G) |f| \times |g| \times 2^{\beta+\delta} \end{aligned} \tag{11}$$

The computational procedure is primarily concerned with the formation of $|f| \times |g| \times 2^{\beta+\delta}$ since $(\text{sign } F \times G)$ can be recorded in advance and used later to apply the correct sign to the product. In addition to recording $(\text{sign } F \times G)$, we record the exponents β and δ after the product $|f| \times |g|$ is formed. The following algorithm is proposed for multiplying F by G :

1. Determine and record $(\text{sign } F \times G)$.
2. Form $|F|$ and $|G|$.
3. Record the leftmost nine bits of $|F|$ and $|G|$. This, in effect, records β and δ .

4. Shift the 23 bits of $|f|$ and $|g|$ left until each has the bit pattern



If this procedure is followed, $|f|$ is no longer represented by (3) during the computation in step 5 below, but has the form

$$|f| = C + D \times 2^{-15} \tag{12}$$

where C lies in the range

$$1/2 \leq C \leq 1 - 2^{-15} \tag{13}$$

and D lies in the range

$$0 \leq D \leq 1 - 2^{-8} \tag{14}$$

Likewise g has the form

$$|g| = A + B \times 2^{-15} \tag{15}$$

where A lies in the range

$$1/2 \leq A \leq 1 - 2^{-15} \tag{16}$$

and where B lies in the range

$$0 \leq B \leq 1 - 2^{-8} \tag{17}$$

5. Use fixed-point operations in forming the product.

$$\begin{aligned} |f| \times |g| &= (C + D \times 2^{-15})(A + B \times 2^{-15}) \\ &= CA + (CB + DA) \times 2^{-15} + DB \times 2^{-30} = DB \times 2^{-30} \\ &= CA + (CB + DA) \times 2^{-15} \end{aligned} \tag{18}$$

Notice that the term $DB \times 2^{-30}$ may be ignored, because once the product is placed back in standard form, only 23 bits are retained. Notice also that (18) is written in such a way that it exhibits the efficiency of the following choice of computational steps:

- a. Form CA giving a double-length product.
- b. Form CB and retain the most significant half of the double-length product.
- c. Form DA.

- d. Add the most significant half of DA to the most significant half of CB.
 - e. Add the least significant half of CA to the sum obtained in (d). This result is the second half of the double-length product. The first half of the double-length product is the most significant half of CA which was formed in a.
6. Next, round and normalize the product obtained using (18) in step 5. Any adjustment in the exponent $\beta + \delta$ which is necessary because of the normalization of $|f| \times |g|$ must be performed.
 7. Finally, pack the 23 bits of the normalized product and the nine bits representing the sign and the adjusted exponent into two 16-bit words (in the standard way). If (sign $F \times G$) is negative, the two words must then be complemented to give the correct sign to the product.

Division

$$\frac{G}{F} = \frac{g \times 2^\delta}{f \times 2^\beta} \tag{19}$$

$$= (\text{sign } G \times F) \left| \frac{g}{f} \right| \times 2^{\delta - \beta}$$

And since the following is wanted:

$$\left| \frac{g}{f} \right| < 1 \tag{20}$$

scale the numerator and write:

$$\frac{G}{F} = (\text{sign } G \times F) \frac{\left| \frac{g}{2} \right|}{|f|} \times 2^{\delta - \beta + 1} \tag{21}$$

Thus, propose the following algorithm for dividing G by F:

1. Determine and record (sign $F \times G$).
2. Form $|F|$ and $|G|$.
3. Record the leftmost nine bits of $|F|$ and $|G|$. This, in effect, records β and δ .
4. Arrange the 23 bits of $|f|$ to give the bit pattern

+	15 most significant bits of $ f $	C
---	-----------------------------------	---

0	8 least significant bits	7 zeros	D
---	--------------------------	---------	---

and the 23 bits of $|g|$ to give the bit pattern

+	0	14 most significant bits of $ g $	A
---	---	-----------------------------------	---

0	9 least significant bits	6 zeros	B
---	--------------------------	---------	---

Thus, f is represented by (12), (13), and (14) as in the case of multiplication. However, in this case

$$\left\lfloor \frac{g}{2} \right\rfloor = A + B \times 2^{-15} \quad (22)$$

where A lies in the range

$$1/4 \leq A \leq 1/2 - 2^{-15} \quad (23)$$

and where B lies in the range

$$0 \leq B \leq 1 - 2^{-9} \quad (24)$$

5. Use fixed-point operations in forming the quotient:

$$\begin{aligned} \left\lfloor \frac{g}{2} \right\rfloor / f &= \frac{A + B \times 2^{-15}}{C + D \times 2^{-15}} \\ &= \left[\frac{A + B \times 2^{-15}}{C} \right] \times \left[\frac{1}{1 + \frac{D}{C} \times 2^{-15}} \right] \\ &= \left[\frac{A}{C} + \frac{B}{C} \times 2^{-15} \right] \times \left[1 - \frac{D}{C} \times 2^{-15} + \frac{D^2}{C^2} \times 2^{-30} + \dots \right] \\ &= \frac{A}{C} + \frac{B}{C} \times 2^{-15} - \frac{AD}{C^2} \times 2^{-15} - \frac{BD}{C^2} \times 2^{-30} + \frac{AD^2}{C^3} \times 2^{-30} + \dots \\ &= \frac{A}{C} + \left[\frac{B}{C} - \frac{AD}{C^2} \right] \times 2^{-15} + \left[\frac{AD^2}{C^3} - \frac{BD}{C^2} \right] \times 2^{-30} + \dots \\ &= \frac{A + \left[B - \frac{AD}{C} \right] \times 2^{-15}}{C} \end{aligned} \quad (25)$$

Notice that the terms beginning with

$$\left[\frac{AD^2}{C^3} - \frac{BD}{C^2} \right] \times 2^{-30}$$

may be ignored because only 23 bits of the quotient are retained. Notice also that (25) is written in such a way that it exhibits the efficiency of the following choice of computational steps.

- a. Form $-AD$ giving a double length product.
- b. Divide $-AD$ (as a double length dividend) by C .

c. Form $B - \frac{AD}{C}$ (rounded to a single length).

d. Form the double-length dividend:

$$A + \left[B - \frac{AD}{C} \right] \times 2^{-15}$$

(The sign of the second term requires special attention.)

e. Divide this double length dividend by C.

f. To obtain the second half of the double-length quotient, the remainder resulting from the division in the previous step must now be divided by C. This procedure is efficient only on those machines which feature fixed-point multiplication that a double-length product which can be used as a double-length dividend for fixed-point division.

6. Next, round and normalize the quotient obtained using (25) and the procedure of step 5. Any adjustment in the exponent which is necessary because of the normalization of

$$\left| \frac{\frac{g}{2}}{F} \right|$$

must be performed.

7. Finally, pack the 23 bits of the normalized quotient and the nine bits representing the sign and the adjusted exponent into two 16-bit words (in the standard way). If (sign $F \times G$) is negative, the two words must then be complemented to give the correct sign to the quotient.

Addition

$$F + G = f \times 2^\beta + g \times 2^\delta \quad (26)$$

The basic problem in floating-point addition is to adjust the exponent of F (or G) so that the binary points are aligned before the addition takes place.

Let L represent a pair of cells which contain the larger of the two numbers F and G, and S represent a pair of cells which contain the smaller of the two numbers. Assume that F is larger than G if

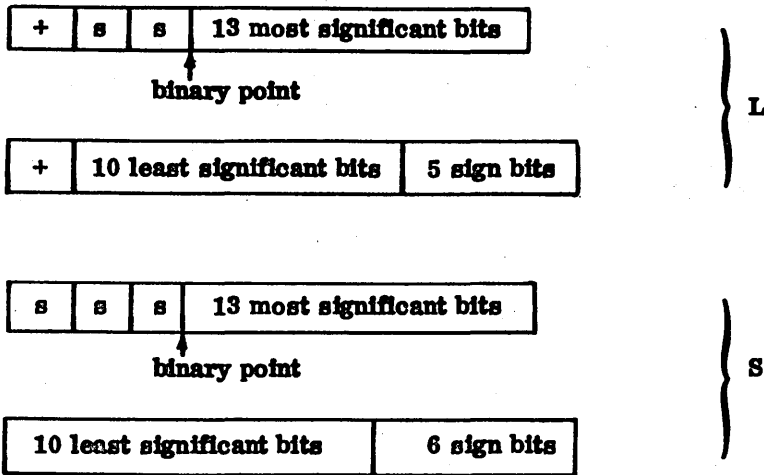
$$\beta \geq \delta \quad (27)$$

and F is smaller than G if

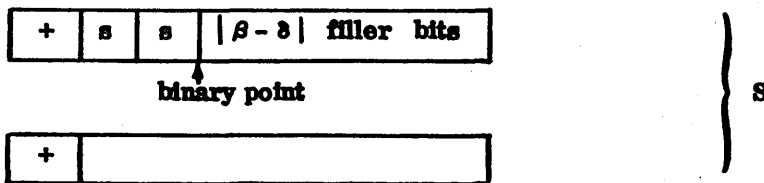
$$\beta < \delta \quad (28)$$

Relative magnitudes of f and g , in case the exponents are equal, are of no concern. Using this convention, process the following algorithm for forming $F + G$.

1. Record the leftmost nine bits of F and G . This, in effect, records β and δ .
2. Determine the sign of $(\beta - \delta)$ and thus determine whether F is smaller or larger than G according to (27) and (28).
3. Place f and g in L and S . If F is larger than G , then f goes into L ; otherwise f goes into S and g goes into L . The following bit patterns should be formed (here s means sign bit).

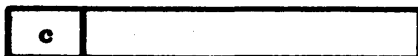


4. Shift S right $|\beta - \delta|$ places and put a + bit at the beginning of each of the two words. If $|\beta - \delta| = 23$, then there is no need to continue since all significant bits in S will be lost.



Notice that the $|\beta - \delta|$ filler bits between the binary point in S and the most significant bit of the fraction are sign bits. This is mathematically correct in a one's complement representation of negative numbers.

5. Add the second halves of L and S .



The first bit of this sum is c . If it is a one, there is actually a carry. However, it usually is easier to add c (refer to step 6) than to test to see whether or not it needs to be added as a carry bit in forming the sum of the first halves of L and S .

6. Add the first halves of L and S and add the carry bit obtained from step 5.

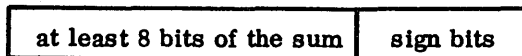
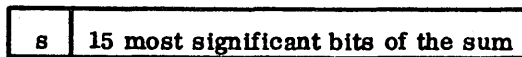


If $e = 1$ then an end-around carry must be performed. This means that a one is added at the right end of the word produced in step 5. Since this might also produce a carry bit, the c in the diagram (refer to step 5) must be cleared to zero before the end-around carry. If a carry bit is again produced, then a one must be added at the right end of the word produced in step 5. It can be shown that this last operation can never produce another $e = 1$.

If $v = s$ then v is a sign bit. If $v \neq s$ then there has been overflow during the addition and v is the most significant bit of the sum. In the latter case, an adjustment of the exponent will be necessary to give the correct answer.

7. Shift the second half of the sum left one place to clear out carry bit c . Then shift the double length sum left (a) one place if $v \neq s$; (b) two places if $v = s$.

This leaves the sum in the following form:



8. If the double-length sum was shifted one place left in step 7 ($v \neq s$) then the exponent must be adjusted to take care of the overflow. This means adding one to the exponent β or δ , whichever is larger. (This will be the exponent of the sum.) If the double-length sum was shifted two places left in step 7, no adjustment of exponent is necessary.
9. The form of the sum given by step 7 must be checked for normalization since it is possible that several of the leading bits of the sum may be zero. (Cancellation occurs when two numbers of opposite sign but nearly equal magnitude are added.) If the sum is not normalized at this point appropriate adjustments in the exponents should be made.
- If 23 left shifts are not sufficient for normalization then the sum should be made zero.
10. At this point the normalized sum may be rounded, although the extra coding involved may not be worth the gain. If rounding is desired, then there are two cases to be considered depending on the sign of the sum. These cases require that care be taken in handling any carry bit produced by the rounding operations.
11. Now pack the 23 most significant bits of the sum, along with nine bits representing the sign and exponent, into two 16-bit words (in the standard way). If the sign of the sum is negative, then the first nine bits must be complemented before the packing takes place.

Subtraction

No special subroutine is necessary since

$$F - G = F + (-G)$$

and one merely complements G before entering the addition subroutine.

FAULT CONDITIONS

If exponent underflow is encountered, a floating-point zero results. If exponent overflow is encountered, the largest word of the appropriate sign results. A divide check is treated as overflow.

REFERENCES

Robert T. Gregory and James L. Raney, "Floating Point Arithmetic with 84-Bit Numbers", Communications of the ACM, Volume 1, Number 1, January 1964.

DOUBLE-PRECISION FLOATING-POINT PACKAGE

D

The software double-precision floating-point package used by FORTRAN is described in this appendix. The package can also be used by an assembly-produced program. For efficiency the package is not run-anywhere.

There are two similar floating-point packages. They are called by the same name (HDFLOT), but one is re-entrant and the other non-re-entrant. Both packages are usable by run-anywhere programs. The re-entrant package must operate in protected core. The non-re-entrant package may operate anywhere.

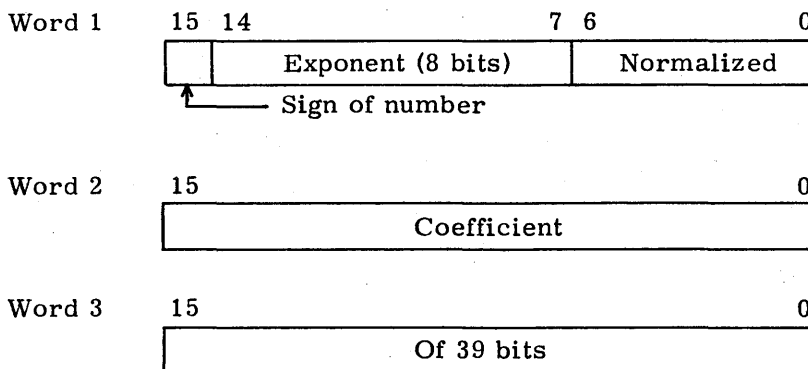
The non-re-entrant version, DFLOTN, utilizes temporary storage to perform its computations. The re-entrant version, DFLOTR, utilizes volatile storage for temporary storage.

If the firmware version of the package is used, the call to HDFLOT is vectored to the firmware where equivalent functions are performed.

NOTE

In some cases, the firmware has approximately two digits less precision than the software package.

Each double-precision floating-point number requires three consecutive words of storage. The first word, containing the most significant bits, is the one that is addressed. Normalized floating-point format is as follows:



Thus the numbers, X, expressible are of the range $-2^{127}(1-2^{-39}) \leq X \leq 2^{127}(1-2^{-39})$ and are significant to one part in 549 billion. If the most significant word is zero (16 bits of zero or 1) a floating-point zero is assumed.

COEFFICIENT

The coefficient consists of a 39-bit number n, $1-2^{-39} \geq n \geq 1/2$. The high-order bit position of the first word is the coefficient sign bit. A 0 denotes a positive coefficient and 1 denotes a negative coefficient. When the coefficient is negative, the entire floating-point number, exclusive of the sign bit, is stored in complement form.

EXPONENT

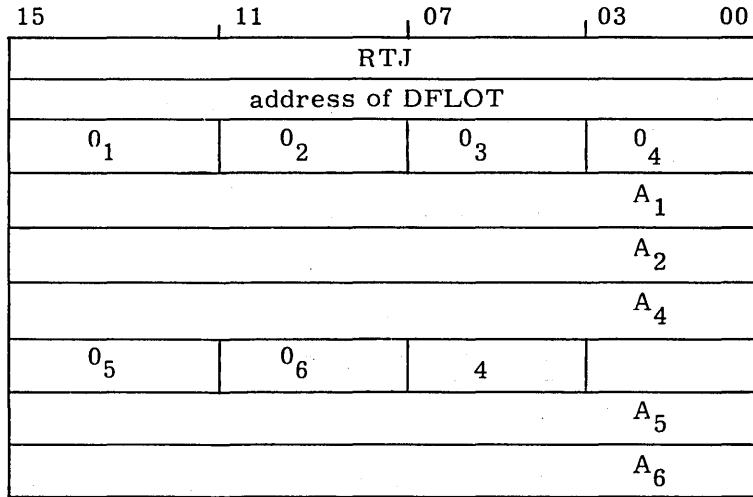
The floating-point exponent is an eight-bit quantity with value ranging from 00 to FF₁₆. Through biasing by 80₁₆, this range expresses both positive and negative exponents.

CALLING SEQUENCE

DFLOT uses an interpretive calling sequence. Both the re-entrant and non-re-entrant calling sequences save the Q, A, and I registers in temporary storage. The interpretation is on a string of four-bit bytes, where the leftmost four-bit byte represents the first operation. Their respective operands, if they exist, follow in the bytes' respective order, one word per byte.

As many bytes may exist as desired, but the last one must be 4.

Example:



User's program resumes

The calling sequence was designed to minimize the amount of core needed, including core used to set up the calling sequence.

OPERATIONS

A description of the following operations and their four-bit byte codes follows.

<u>Operation</u>	<u>4-Bit Code</u>	<u>Meaning</u>
SPEC	0	Special. This is first byte of a two-byte operation code.
FLOF	1	Float to fix. The pseudo accumulator is converted to a 16-bit integer and stored at the effective operand address.
FIXF	2	Fix to float. The contents of the effective operand address are converted from a 16-bit integer number and placed in the pseudo accumulator.
STRI	3	Store index. The index register is stored at the effective operand address.
FEND	4	End of calling sequence. This operation terminates the calling sequence. No operand needed.
CHMD	5	Change mode of operation. All operand addresses following this operation code in the calling sequence are made relative if the preceding addresses were absolute, or absolute if the preceding addresses were relative. Addresses are initially absolute. No operand is needed.
NIDX	6	No index. The succeeding operands do not have indexing increments. NIDX supersedes any preceding INDX and is superseded by any following INDX. NIDX is assumed initially. No operand is needed.
FCOM	7	Floating complement. The pseudo accumulator is complemented. No operand is needed.
FSUB	8	Floating subtract. The contents of the effective operand address are subtracted from the pseudo accumulator and the result is left in the pseudo accumulator.
FMPY	9	Floating multiply. The pseudo accumulator is multiplied by the contents of the effective operand address and the result is left in the pseudo accumulator.
FDIV	A ₁₆	Floating divide. The pseudo accumulator is divided by the contents of the effective operand address and the result is left in the pseudo accumulator.
FLDD	B ₁₆	Floating load. The floating-point number in the corresponding effective operand address is transferred to the pseudo accumulator.
ADDI	C ₁₆	Add to index. The contents of the effective operand address are added to the index register.
FLST	D ₁₆	Floating store. The floating-point number is transferred from the pseudo accumulator to the corresponding effective operand address.
FADD	E ₁₆	Floating add. The contents of the effective operand address are added to the pseudo accumulator, and the sum is left in the pseudo accumulator.
INDX	F ₁₆	Index. The operand corresponding to INDX is used to increment the operand of the following operations: FLDD, FLST, FADD, FSUB, FMPY, and FDIV. Each succeeding INDX supersedes the last. No index is initially assumed.

The following operation codes are executed only if the preceding byte is a SPEC(0).

<u>Operation</u>	<u>4-Bit Code</u>	<u>Meaning</u>
CACS	1	Continue another calling sequence. The next operation to be executed is the most significant four-bit command code located at the effective operand address (unconditional branch).
BRAM	2	Branch accumulator minus. If the pseudo accumulator is minus, control is transferred to the effective operand address.
BRAZ	3	Branch accumulator zero
BRAN	4	Branch accumulator nonzero
BRAP	5	Branch accumulator positive
BRIM	6	Branch index register minus
BRIZ	7	Branch index register zero
BRIN	8	Branch index register nonzero
BRIP	9	Branch index register positive

NOTE

Codes 0 and A₁₆ minus F₁₆, when preceded by a SPEC code, are executed as FEND.

RELATIVE ADDRESSING

The operand address, bits 15 through 0, is relative to self. The relative address is computed by subtracting the calling sequence operand address from the actual operand address.

Example:

$$X = - (A(I)+B(I))*C(I)+D(J)*E(J)$$

Assume TEMP, X, J, D, and E are absolutely addressed, and the other operands are relatively addressed. The call to DFLOT would look like the following.

RTJ		DFLOT	
F ₁₆	B ₁₆	9	6
Absolute address of J			
Absolute address of D			
Absolute address of E			
D ₁₆	5	F ₁₆	B ₁₆
Absolute address of temporary cell TEMP			
Relative address of I			
Relative address of A			
E ₁₆	9	7	6
Relative address of B			
Relative address of C			
5	E ₁₆	D ₁₆	4
Absolute address of temporary cell TEMP			
Absolute address of X			

FAULT CONDITIONS

At any time during execution, the following fault conditions are flagged: exponent overflow, exponent underflow, and divide fault (attempted division by zero). These fault conditions may be tested and reset by use of the IFALT function call.

FLOATING-POINT ARITHMETIC WITH 39-BIT NUMBERS

A classic and straightforward technique is presented which is not limited to the size or type of the number representation used.

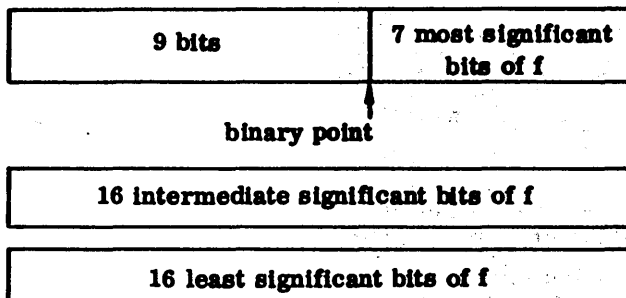
Consider the double-precision floating-point number:

$$F = f \times 2^\beta \tag{1}$$

where $|f|$ lies in the range

$$1/2 \leq |f| \leq 1 - 2^{-39} \tag{2}$$

Assume that we have a machine with a word length of 16 bits and that the 48 bits in the triple-length word are divided in the following standard way:



The leftmost block of nine bits is divided into three parts:

- The first (leftmost) bit represents the sign of f .
- The second bit represents the sign of β .
- The next seven bits represent the magnitude of β .

This allows 39 bits for the representation of f . We shall assume that the binary point lies at the left of the 39 bits representing f so that the seven most significant bits of f are stored in the first word of the three, and the 16 least significant bits of f are stored in the third word.

If we write

$$|f| = c + c_i \times 2^{-7} + d \times 2^{-23} \tag{3}$$

where c lies in the range

$$1/2 \leq c \leq 1 - 2^{-7} \tag{4}$$

c_i lies in the range

$$0 \leq c_i \leq 1 - 2^{-16} \tag{5}$$

and where d lies in the range

$$0 \leq d \leq 1 - 2^{-16} \tag{6}$$

then c represents the seven most significant bits of f , c_i represents the 16 intermediate bits of f , and d represents the 16 least significant bits of f .

FOUR ARITHMETIC OPERATIONS

We wish to consider the four basic arithmetic operations using double-precision floating-point numbers of the form discussed. Consequently, in order to have notation for two operands, let us consider a second double-precision floating-point number.

$$G = |g| \times 2^5 \quad (7)$$

where $|g|$ lies in the range

$$1/2 \leq |g| \leq 1 - 2^{-39} \quad (8)$$

If

$$|g| = a + a_1 \times 2^{-7} + b \times 2^{-23} \quad (9)$$

where a lies in the range

$$1/2 \leq a \leq 1 - 2^{-7} \quad (10)$$

a_1 lies in the range

$$0 \leq a_1 \leq 1 - 2^{-16} \quad (11)$$

and b lies in the range

$$0 \leq b \leq 1 - 2^{-16} \quad (12)$$

then a represents the seven most significant bits of $|g|$, a_1 represents the 16 intermediate significant bits of $|g|$, and b represents the 16 least significant bits of $|g|$.

Assume that the machine represents negative numbers using a one's complement system. Assume that the procedure for changing the sign of a double-precision floating-point number is to perform a bit-by-bit complement of the entire 48 bits (including the nine bits representing the sign and exponent).

Addition

$$F + G = f \times 2^\beta + g \times 2^\delta \quad (13)$$

The basic problem in floating-point addition is to adjust the exponent of F (or G) so that the binary points are aligned before the addition takes place.

Let L represent three cells which contain the larger of the two numbers F and G , and S represent three cells which contain the smaller of the two numbers. Assume that F is larger than G if

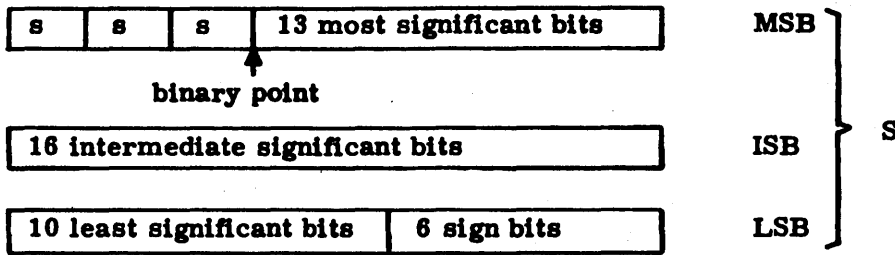
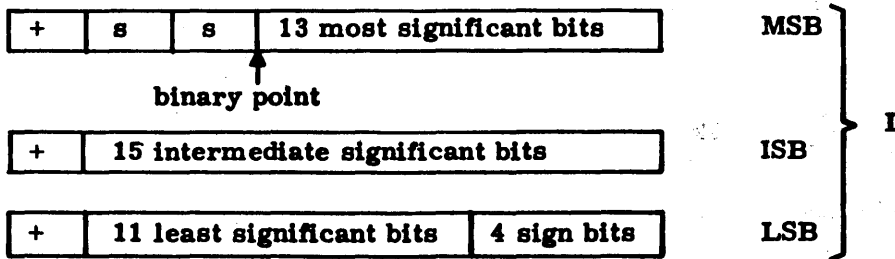
$$\beta \geq \delta \quad (14)$$

and F is smaller than G if

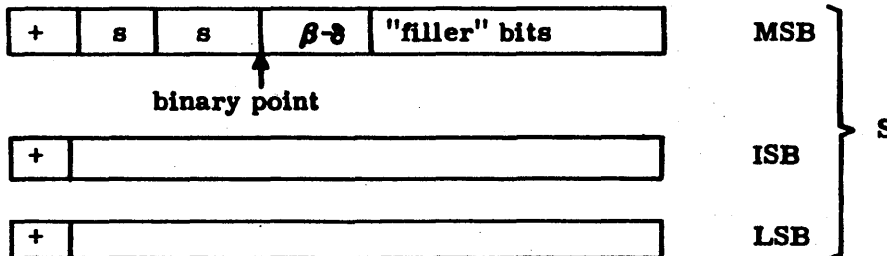
$$\beta < \delta \quad (15)$$

The relative magnitudes of f and g , in case the exponents are equal, are of no concern. Using this convention, process the following algorithm for forming $F + G$:

1. Record the leftmost nine bits of F and G . This, in effect, records β and δ .
2. Determine the sign of $(\beta - \delta)$ and thus determine whether F is smaller or larger than G according to (14) and (15).
3. Place f and g in L and S . If F is larger than G , then f goes into L ; otherwise, f goes into S and g goes into L . The following bit patterns should be formed (here s means sign bit):

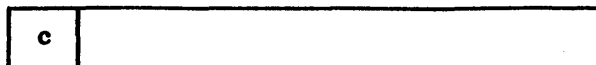


4. Shift S right $|\beta - \delta|$ places and put a $+$ bit at the beginning of each of the three words. If $|\beta - \delta| = 39$, then there is no need to continue since all significant bits in S will be lost.



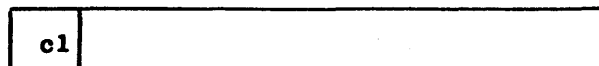
Notice that the $|\beta - \delta|$ filler bits between the binary point in S and the most significant bit of the function are sign bits. This is mathematically correct in a one's complement representation of negative numbers.

5. Add the LSB portions of L and S.



The first bit of this sum is c. If it is a one, there is actually a carry. However, it usually is easier to add c (see step 6) than to test to see whether or not it needs to be added as a carry bit in forming the sum of the LSB portions of L and S.

6. Add the ISB portions of L and S and the carry bit from step 5.



If C1 is set to one, we have a carry and we will add c1 to step 7 in forming the sum of the most significant bits of L and S.

7. Add the MSB portions of L and S and add the carry bit obtained from step 6.

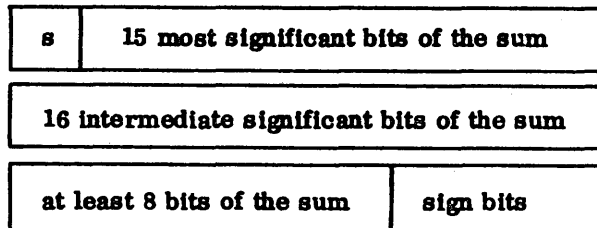


If $e = 1$ then an end-around carry must be performed. This means that a one is added at the right end of the word produced in step 5. Since this might also produce a carry bit, the c in the diagram (see step 5) must be cleared to zero before the end-around carry. If a carry bit is again produced, then a one must be added at the right end of the word produced in step 6. Since this might also produce a carry bit, the c1 in the diagram (see step 6) must be cleared to zero before the end-around carry. If a carry bit is again produced, then a one must be added at the right end of the word above. It can be shown that this last operation can never produce another $e = 1$.

If $v = s$ then v is a sign bit. If $v \neq s$ then there has been overflow during the addition and v is the most significant bit of the sum. In the latter case, an adjustment of the exponent will be necessary to give the correct answer.

8. Shift the LSB portion of the sum left one place to clear out carry bit c. Then shift the ISB portion of the sum left one place to clear out the carry bit c1. Then shift the LSB portion of the sum one place and put the bit shifted off into the rightmost bit of the ISB portion of the sum. Then shift the triple length sum left (a) one place if $v \neq s$; (b) two places if $v = s$.

This leaves the sum in the following form:



9. If the triple-length sum was shifted one place left in step 8 ($v \neq s$) then the exponent must be adjusted to take care of the overflow. This means adding one to the exponent β or δ , whichever is larger. (This will be the exponent of the sum.) If the double length sum was shifted two places left in step 8, no adjustment of exponent is necessary.
10. The form of the sum given by step eight must be checked for normalization since it is possible that several of the leading bits of the sum may be zero. (Cancellation occurs when two numbers of opposite sign but nearly equal magnitude are added.) If the sum is not normalized at this point, appropriate adjustments in the exponents should be made.
If 39 left shifts are not sufficient for normalization, then the sum should be made zero.
11. At this point the normalized sum may be rounded, although the extra coding involved may not be worth the gain. If rounding is desired, then there are two cases to be considered depending on the sign of the sum. These cases require that care be taken in handling any carry bit produced by the rounding operations.
12. Now pack the 39 most significant bits of the sum, along with nine bits representing the sign and exponent, into three 16-bit words (in the standard way). If the sign of the sum is negative, then the first nine bits must be complemented before the packing takes place.

Subtraction

No special subroutine is necessary since

$$F - G = F + (-G)$$

and one merely complements G before entering the addition subroutine.

Multiplication

$$\begin{aligned} F \times G &= (f \times 2^\beta) (g \times 2^\delta) && (16) \\ &= (\text{sign } F \times G) |f| \times |g| \times 2^{\beta+\delta} \end{aligned}$$

The computational procedure is primarily concerned with the formation of $|f| \times |g| \times 2^{\beta+\delta}$ since $(\text{sign } F \times G)$ can be recorded in advance and used later to apply the correct sign to the product. In addition to recording $(\text{sign } F \times G)$, we record the exponents β and δ after the product $|f| \times |g|$ is formed. The following algorithm is proposed for multiplying F by G:

1. Determine and record $(\text{sign } F \times G)$
2. Form $|F|$ and $|G|$.
3. Record the leftmost nine bits of $|F|$ and $|G|$. This, in effect, records β and δ .

4. Shift the 39 bits of $|f|$ and $|g|$ left until each has the bit pattern

+	15 most significant bits	C and A
0	15 intermediate significant bits	C _i and A _i
0	9 least significant bits	D and B
	6 zeros	

If this procedure is followed, $|f|$ is no longer represented by (3) during the computation in step 5 below, but has the form:

$$f = C + C_i \times 2^{-15} + D \times 2^{-30} \quad (17)$$

where C, C_i, and D lie in the following ranges:

$$2^{-1} \leq C \leq 1 - 2^{-15} \quad (18)$$

$$0 \leq C_i \leq 1 - 2^{-15} \quad (19)$$

$$0 \leq D \leq 1 - 2^{-9} \quad (20)$$

Likewise $|g|$ has the form

$$|g| = A + A_i \times 2^{-15} + B \times 2^{-30} \quad (21)$$

where A, A_i, and B lie in the following ranges:

$$2^{-1} \leq A \leq 1 - 2^{-15} \quad (22)$$

$$0 \leq A_i \leq 1 - 2^{-15} \quad (23)$$

$$0 \leq B \leq 1 - 2^{-9} \quad (24)$$

5. Use fixed-point operations in forming the product.

$$\begin{aligned} |f| \times |g| &= (C + C_i \times 2^{-15} + D \times 2^{-30}) (A + A_i \times 2^{-15} + B \times 2^{-30}) \\ &= CA + (CA_i + C_iA) \times 2^{-15} + (DA + C_iA_i + CB) \times 2^{-30} \\ &\quad + (DA_i + C_iB) \times 2^{-45} + DB \times 2^{-60} \\ &= CA + (CA_i + C_iA) \times 2^{-15} + (DA + C_iA_i + CB) \times 2^{-30} \end{aligned} \quad (25)$$

Notice that the terms $(DAI + CIB) \times 2^{-45}$ and $DB \times 2^{-60}$ may be ignored, because once the product is placed back in standard form, only 39 bits are retained. The following computational steps are performed:

- a. Form CA giving a double-length product.
 - b. Form DA and retain the most significant half of the double-length product.
 - c. Form CiAi and retain the most significant half of the double-length product.
 - d. Form CiA giving a double-length product.
 - e. Add the most significant half of DA to the most significant half of CiAi.
 - f. Form CB and retain the most significant half of the double-length product.
 - g. Add the most significant half of DB to the sum obtained in (e).
 - h. Add the least significant half of CiA to the sum obtained in (g).
 - i. Form CAi giving a double-length product.
 - j. Add the least significant half of CAi to the sum obtained in (h). This result is the least significant portion of the triple-length product.
 - k. Add the most significant half of CiA to CAi.
 - l. Add the least significant half of CA to the sum obtained in (k). This result is the intermediate significant portion of the triple-length product. The first half of the double-length product is the most significant half of CA which was formed above in (a).
6. Next, round and normalize the product obtained using (25) in step 5. Any adjustment in the exponent $\beta + \delta$ which is necessary because of the normalization of $|f| \times |g|$ must be performed.
 7. Finally, pack the 39 bits of the normalized product and the nine bits representing the sign and the adjusted exponent into three 16-bit words (in the standard way). If the (sign $F \times G$) is negative, the two words must then be complemented to give the correct sign to the product.

Division

$$\begin{aligned} \frac{G}{F} &= G \times \frac{1}{F} = (g \times 2^\delta) \times \left(\frac{1}{f \times 2^\beta} \right) \\ &= (\text{sign } G \times F) \times g \times \frac{1}{f} \times 2^{\delta - \beta} \end{aligned} \quad (26)$$

As a matter of fact, since we want:

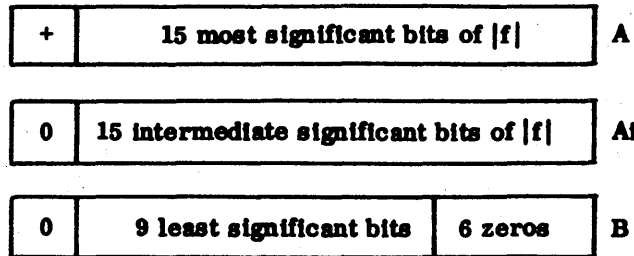
$$\left| \frac{g}{f} \right| < 1, \quad (27)$$

scale the numerator and write:

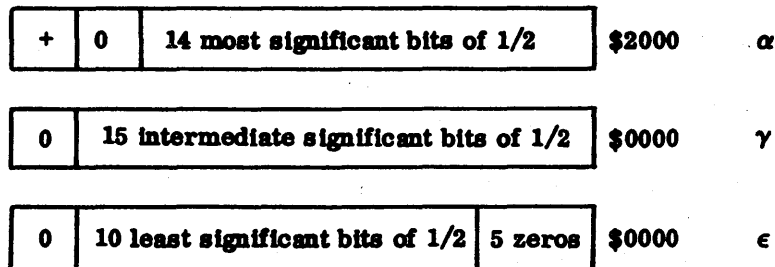
$$\frac{G}{F} = (\text{sign } G \times F) \times |g| \times \frac{\left| \frac{1}{2} \right|}{f} \times 2^{\delta - \beta + 1} \quad (28)$$

Thus, propose the following algorithm for dividing G by F:

1. Determine and record (sign F × G).
2. Form |F| and |G|.
3. Record the leftmost eight bits of |F| and |G|. This, in effect, records β and δ.
4. Arrange the 39 bits of |f| to give the bit pattern:



and the 39 bits which represent the number 1.0 to give the bit pattern:



5. Use fixed-point operations in forming the quotient.

$$\frac{1}{A + A_1 \times 2^{-15} + B \times 2^{-30}} = \frac{\alpha + \gamma \times 2^{-15} + \epsilon \times 2^{-30}}{A + A_1 \times 2^{-15} + B \times 2^{-30}}$$

where: α = \$2000
 γ = \$0000
 ε = \$0000

$$\begin{aligned} &= \frac{\alpha}{A} + \frac{1}{A} \left(\gamma - \frac{\alpha A_1}{A} \right) \times 2^{-15} + \frac{1}{A} \left(\epsilon - \frac{\alpha B}{A} \right) \times 2^{-30} \\ &\quad - \frac{A_1}{A^2} \left(\gamma - \frac{\alpha A_1}{A} \right) \times 2^{-30} \\ &= \frac{1}{A} \left[\alpha - \frac{\alpha}{A} [A_1 \times 2^{-15} + (B - \frac{A_1^2}{A}) \times 2^{-30}] \right] \end{aligned} \quad (29)$$

Any terms beginning with 2^{-45} are ignored because only 39 bits of the quotient are retained. The following computational steps are performed:

- a. Form $-A1^2$ giving a double-length product.
- b. Divide $-A1^2$ (as a double-length dividend) by A.
- c. Form $B - \frac{A1^2}{A}$ (rounded to a single length).
- d. Form the double length-dividend:

$$A1 \times 2^{-15} + \left[\left(B - \frac{A1^2}{A} \right) \times 2^{-30} \right]$$

(The sign of the second term requires special attention.)

- e. Divide the double-length dividend by A and multiply the result by α . The multiply is accomplished by shifting the result of the divide.
- f. To obtain the second half of the double-length quotient, the remainder resulting from the division in the previous step must now be divided by A.
- g. Form α (the most significant bits of $1/2$) and the result obtained from step e as a double-length dividend.
- h. Divide the double-length dividend by A. The result is the most significant bits of the quotient.
- i. Form the remainder of step h and the result of step f as a double-length dividend.
- j. Divide the double-length dividend by A. The result is the intermediate significant bits of the quotient.
- k. Divide the remainder obtained in step j by A. The result is the least significant bits of the quotient.
- l. Next, round and normalize the three-word quotient using (29) and the procedure of step 5. Any adjustment in the exponent which is necessary because of normalization of

$$\frac{1}{2^f}$$

must be performed.

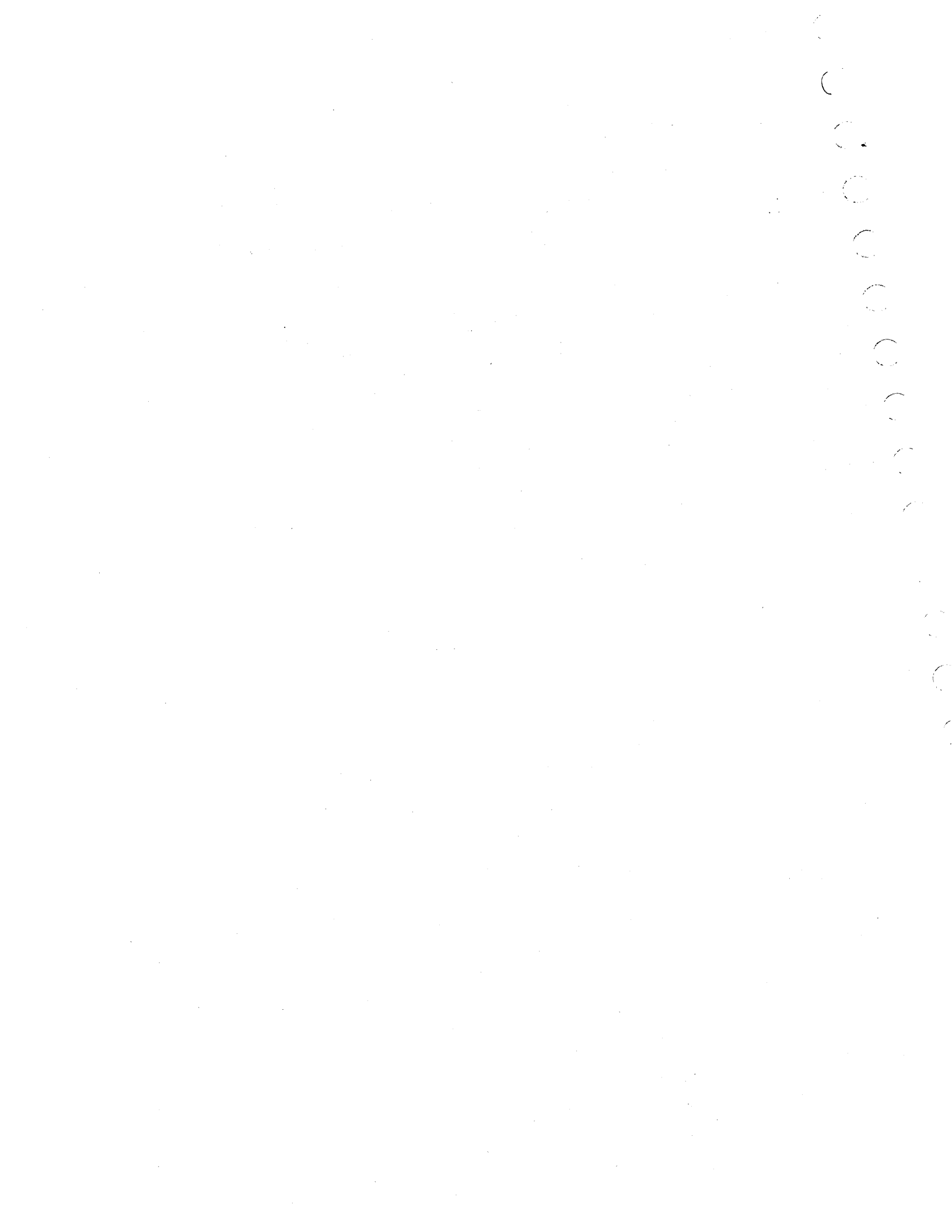
- m. The three-word quotient is then multiplied by $|g|$.
- n. Next, round and normalize the product. Any adjustment in the exponent $\beta + \delta$ which is necessary due to the normalization must be performed.
- o. Finally, pack the 39 bits of the normalized quotient and the nine bits representing the sign and the exponent into three 16-bit words (in the standard way). If (sign $F \times G$) is negative, the three words must then be complemented to give the correct sign to the quotient.

FAULT CONDITIONS

If exponent underflow is encountered, a floating point zero results. If exponent overflow is encountered, the largest word of the appropriate sign results. A divide check is treated like an overflow.

REFERENCES

Robert T. Gregory and James L. Raney, "Floating Point Arithmetic with 84-Bit Numbers", Communications of the ACM, Volume 1, Number 1, January 1964.



CODING HINTS

E

The constants in an arithmetic expression should be collected. For example,

$X = Y + 3.1 * 4.2$

should be written

$X = Y + 13.02$

Subexpressions, including a byte variable as constant in a DO loop, should be pulled out of the loop. For example, the program

```
SUBROUTINE SUM
COMMON A(10), IC(10)
DATA B/3.4/
DO 1 I = 1, 10
  A(I) = 0.0
DO 1 J = 1, 10
1  A(I) = (SIN(B) + FLOAT(I)) * FLOAT(IC(J)) + A(I)
RETURN
END
```

should be written

```
SUBROUTINE SUM
COMMON A(10), IC(10)
DATA B/3.4/
TEMP1 = SIN(B)
DO 2 I = 1, 10
  C = 0.0
  TEMP2 = FLOAT(I) + TEMP1
DO 1 J = 1, 10
1  C = TEMP2*FLOAT(IC(J)) + C
2  A(I) = C
RETURN
END
```

Only one dimensional array should be used. If two or three dimensions are desired, the programmer should use the subscript functions given in Appendix G. For example, the program

```
SUBROUTINE TRANSF
COMMON A(10, 10), B(10, 10)
DO 1 J = 1, 10
DO 1 I = 1, 10
1  A(I, J) = B(I, J) + 1.0
RETURN
END
```

should be written

```
SUBROUTINE TRANSF
COMMON A(100), B(100)
DO 1 I = 1, 100
1  A(I) = B(I) + 1.0
RETURN
END
```

Common subexpressions between two or more arithmetic expressions should be collected. For example,

```
Y(I) = A+B + FUNC1(IBYTE)
Z(I) = A+FUNC1(IBYTE) + FUNC2(IBYTE)
```

where IBYTE is a byte variable, should be written

```
ITEMP1 = IBYTE
TEMP1 = A + FUNC1(ITEMP1)
Y(I) = B + TEMP1
Z(I) = TEMP1 + FUNC2(ITEMP1)
```

When a program references a multi-dimensional array, the FORTRAN compiler on occasion generates a relocatable base address for an indexed variable which is intended to fall in front of data, common, or the program. Since this relocatable address is expressed in 15 bits, the loader on a 16-bit load has no way of knowing that this is not a forward relocation. To accommodate this, the loader assumes that any relocatable address in the range 7F80 to 7FFF is intended as backward relocation. This range can be changed by reassembly of the MSOS loader module RBDBZ1.

The user who has the double precision capability may write programs which require only single precision. To avoid linkage to the double precision library, the external references to DOUT, Q8DXP1, and Q8DXP9 must be satisfied. The user may write his own dummy routine with these references as entry points or use the routine DBLDMY (deck ID K19) contained in the MS FORTRAN product set, and load it with his programs. Refer to Section 2.2.6, Double Precision Type Data, and note the evaluation for double precision constants to avoid an external reference to DFLOT.

HARDWARE REQUIREMENTS

F

1. The minimum hardware configuration is:

FORTRAN 3. 2A

Mass Memory Device
(.5 million words or more)
Card Reader
Teletypewriter
CDC CYBER 18-Class CPU
Core Storage Increments*

Compiler core requirement is less
than 9,300 words.

Minimum MSOS Operating System
core requirement is 9.2K

FORTRAN 3. 2B

Mass Memory Device
(.5 million words or more)
Card Reader
Teletypewriter
CDC CYBER 18-Class CPU
Core Storage Increments*

Compiler core requirement is less
than 16,900 words.

Minimum MSOS Operating System
core requirement is 9.2K.

2. The typical configuration is:

FORTRAN 3. 2A

Mass Memory Device
(1.0 million words or more)
Teletypewriter
Card Reader/Punch
Magnetic Tape Devices
CDC CYBER 18-Class CPU
Core Storage Increments*

Compiler core requirement is less
than 9,300 words.

Typical MSOS Operating System
core requirement 19K**.

FORTRAN 3. 2B

Mass Memory Device
(1.0 million words or more)
Teletypewriter
Card Reader/Punch
Magnetic Tape Devices
CDC CYBER 18-Class CPU
Core Storage Increments*

Compiler core requirement is less
than 16,900 words.

Typical MSOS Operating System
core requirement 12.5K**.

*Core requirements are based on the size of the Compiler used and the size of the MSOS operating system configured.

**Typical operating system core requirements for the A variant versus the B variant are different because certain nice-to-have MSOS features are usually not included in the 3.2B system to allow 32K configuration.



ARRAYS

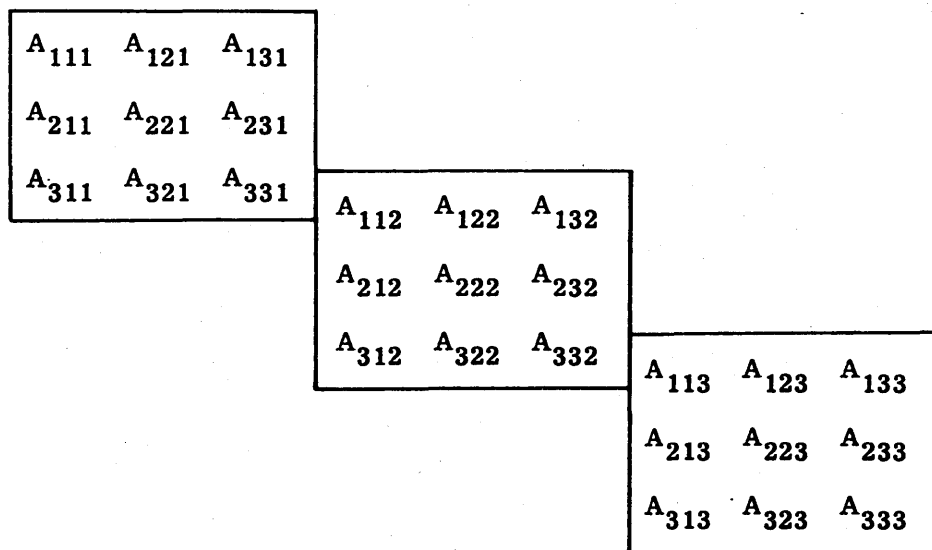
G

An array is a block of sequential memory locations referenced by a single name. The name types the elements of the array as integer or real (Section 2.4). Arrays are dimensioned in the mathematical sense of having rows, columns, and planes. The magnitude of these dimensions is defined by the array declarator, which is the array name followed by a set of numerical subscripts giving the maximum dimensions.

Examples:

IOTA (50)	One-dimensional array with 50 integer elements
BETA (4, 6)	Two-dimensional array with 24 real elements
ALPHA (4, 3, 5)	Three-dimensional array with 60 real elements

Elements of arrays are stored by columns in ascending order of location. The ordering of elements in an array follows the rule that the first subscript varies most rapidly and the last subscript varies least rapidly. In the array declared as A(3, 3, 3)



The planes are stored in order, starting with the first, as follows.

$$\begin{aligned} &A_{111} \overrightarrow{L} \quad A_{121} \overrightarrow{L+3} \dots A_{131} \overrightarrow{L+24} \\ &A_{211} \overrightarrow{L+1} \quad A_{221} \overrightarrow{L+4} \dots A_{231} \overrightarrow{L+25} \\ &A_{311} \overrightarrow{L+2} \quad A_{321} \overrightarrow{L+5} \dots A_{331} \overrightarrow{L+26} \end{aligned}$$

For a given dimensionality, subscript declarator, and subscript, the value of a subscript pointing to an array element and the maximum value a subscript may attain is indicated in Table G-1. A subscript expression must be greater than zero.

The value of the array element successor function is obtained by adding one to the the entry in the subscript value column. Any array element whose subscript has this value is the successor to the original element. The last element of the array is the one whose subscript value is the maximum subscript value and which has no successor element.

The sequential location of a particular element of a stored array is determined according to the following:

Given the array defined by the declarator

AZ(A, B, C)

The ordinal location of element AZ(a, b, c) is given by the formula

$$a + A * (b-1) + A * B * (c-1)$$

Derivation of the formula is illustrated in Figure G-1.

Example:

To find the ordinal location of element B(2, 3, 4) in the array B(5, 6, 7)

$$2 + 5 * (3-1) + 5 * 6 * (4-1) = 102$$

A subscript never may be less than 1 or greater than the maximum dimension declared for it. The elements of one-dimension array BETA (I) may not be referred to as BETA (I, J) or BETA (I, J, K). A diagnostic is issued if this is attempted.

The array name without subscripts references the entire array when it is used in an I/O list, as an argument of a function or subroutine (Sections 7.4.1 and 7.4.2), or in a specification statement other than DIMENSION (Section 6.1.1) or DATA (Section 6.2). The array name without subscripts references the first element when it is used in an expression.

Table G-1. Value of a Subscript

DIMENSIONALITY	SUBSCRIPT DECLARATOR	SUBSCRIPT REFERENCE	SUBSCRIPT VALUE	MAXIMUM SUBSCRIPT VALUE
1	(A)	(a)	a	A
2	(A, B)	(a, b)	a+A*(b-1)	A*B
3	(A, B, C)	(a, b, c)	a+A*(b-1) +A*B*(c-1)	A*B*C

Notes: (1) a, b, and c are subscript expressions.
(2) A, B, and C are dimensions.

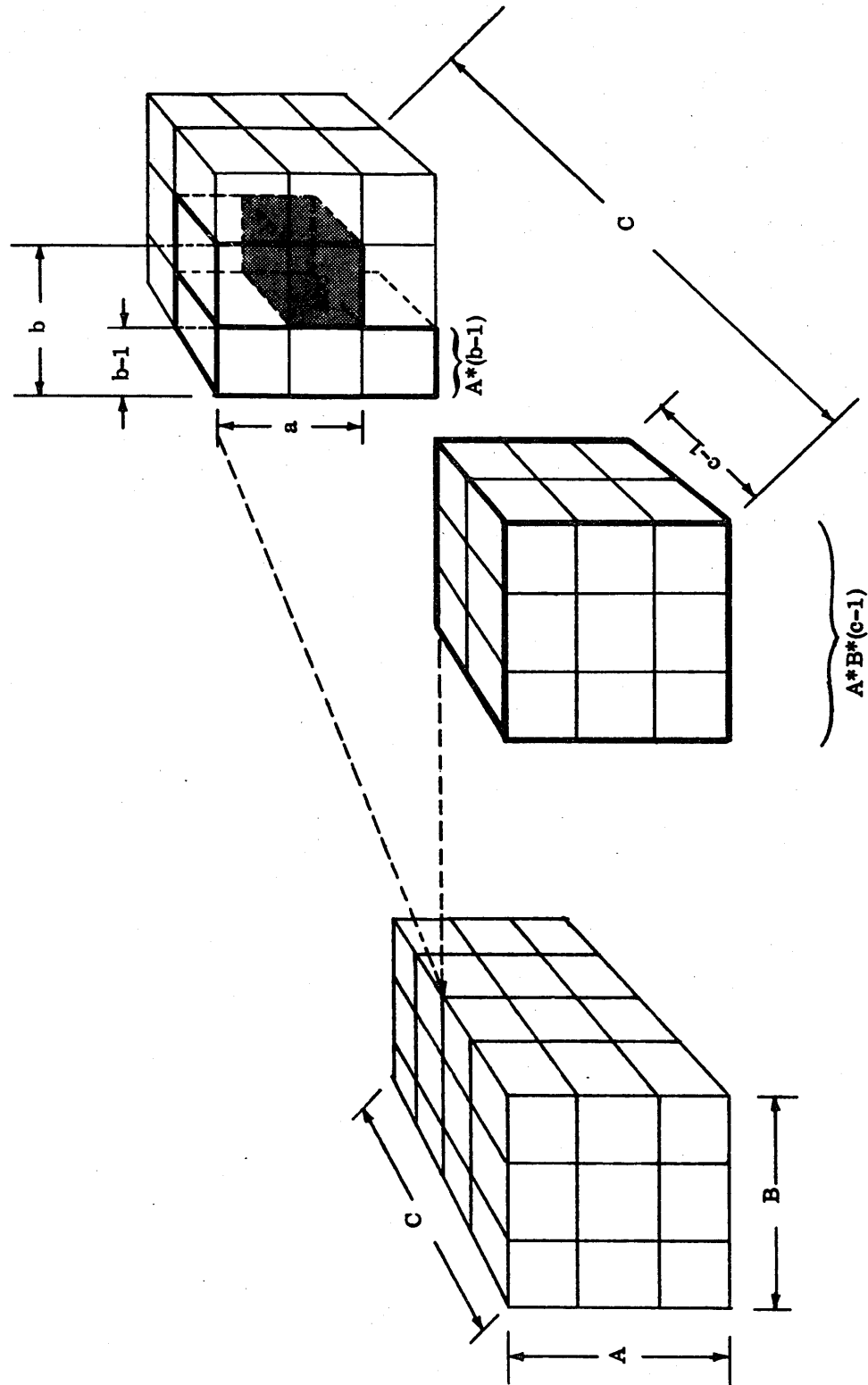


Figure G-1. Array Successor Function $a + A * (b-1) + A * B * (c-1)$

Before an array can be used in a program, its name and dimensionality must be declared in a DIMENSION, COMMON, or type statement (Sections 6.1.1, 6.1.2, and 6.1.4).

Example:

Given the array ALPHA(3,4,4)

It will be declared for program use by any of the following:

DIMENSION ALPHA (3,4,4)

COMMON // ALPHA (3,4,4)

INTEGER ALPHA (3,4,4)

ASCII CODES

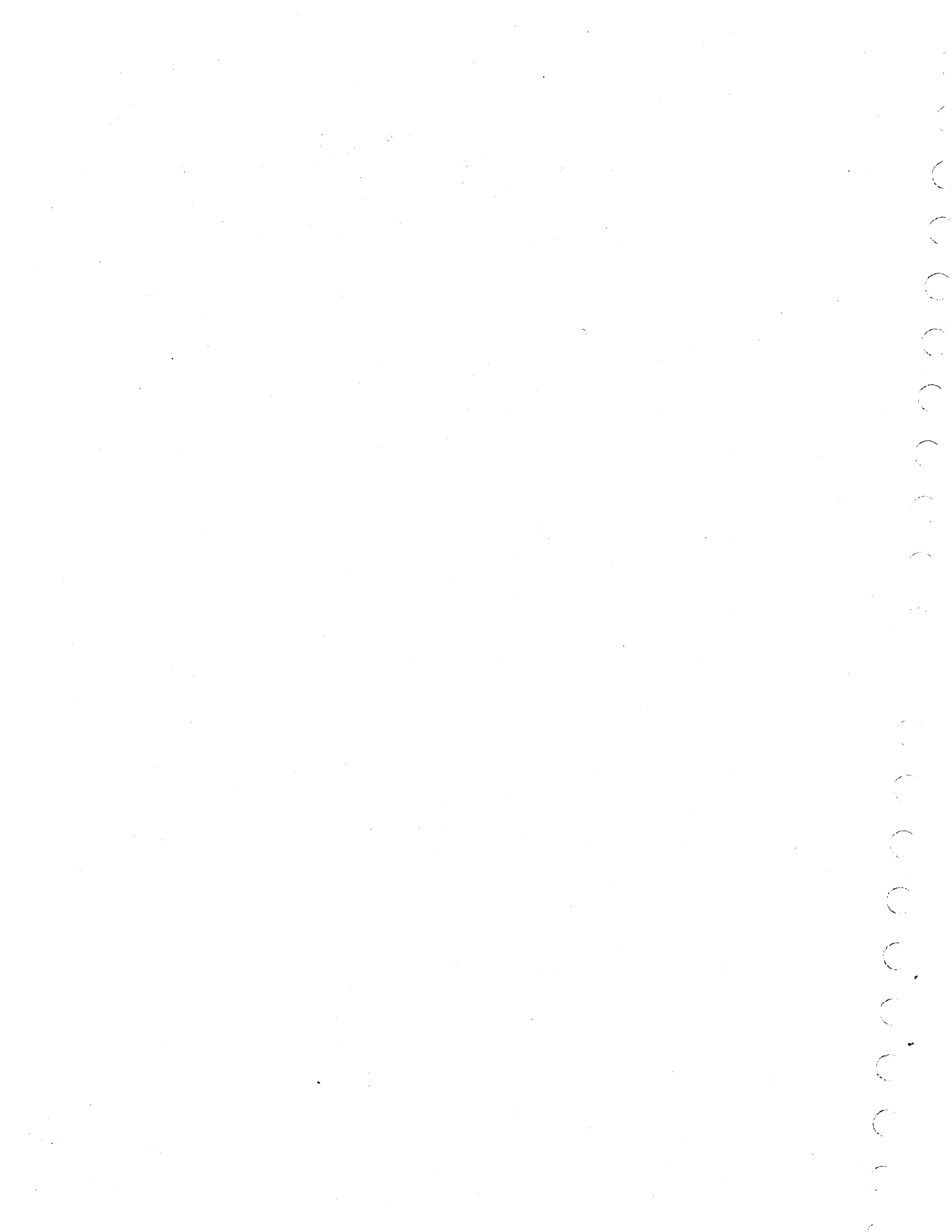
H

The 1968 American Standard Code for Information Interchange (ASCII) is used by MSOS for communication between the CYBER 18/1700 and external I/O devices. In addition to the code for the FORTRAN character set, it includes code for control of the paper tape punch and the teletypewriter.

ASCII code uses eight bits, the first of which is always 0; it is omitted in the following table. Bits 1 through 4 contain the low-order four bits of code for the character in that row. Bits 5, 6, and 7 contain the high-order three bits of the code for the character in that column.

BITS				$\begin{matrix} \xrightarrow{b_7} \\ \xrightarrow{b_6} \\ \xrightarrow{b_5} \end{matrix}$ COLUMN ROW	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓		0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(8	H	X	h	x
1	0	0	1	9	HT	EM)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	
1	1	0	1	13	CR	GS	-	=	M]	m	}
1	1	1	0	14	SO	RS	.	>	N	~	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

FORTRAN Character Set



OPTIMIZATIONS

J

MS FORTRAN optimizations are listed as follows:

1. Index registers are optimally assigned.
2. Relative addressing is used where possible.
3. Storage is allocated to maximize relative addressing. For example, some arrays are put into the middle of code and constants may be duplicated.
4. All simple FORTRAN-provided functions are inserted in-line (for example, IABS or AND).
5. A comprehensive analysis of IF statements is made. Code generated takes cognizance of a transfer from the IF to the label of the next statement; and also if the statement is a GO TO. In a logical IF, the computations are structured to produce the least amount of computation for a determination of the expression's truth value.
6. Arithmetic expressions are analyzed and computed in an order which minimizes both the amount of code generated and its execution time.
7. The compiler may reference the values in A, Q, and I (FF_{16}) and make use of them. It may even reference each of these values by two different names. For example, if $I = 0$, the compiler can reference both I and 0 as representing a value in the accumulator.

1941

Dear Mr. [Name],

I have your letter of [Date] regarding [Subject].

I am sorry that I cannot give you a more definite answer at this time.

The matter is still under consideration and I will contact you again as soon as a final decision has been reached.

Very truly yours,

[Signature]

FORTRAN CHARACTER SET

K

FORTRAN uses alphanumeric and special characters.

Alphanumeric characters are the letters A through Z and digits 0 through 9.

The decimal system is used unless indicated otherwise; however, octal and hexadecimal numbers may be used in certain instances.

Following is a list of characters.

<u>ASCII Character</u>	<u>ASCII Code</u>	<u>Hollerith Punch (026)</u>	<u>Description</u>
0	30	0	Digit
1	31	1	Digit
2	32	2	Digit
3	33	3	Digit
4	34	4	Digit
5	35	5	Digit
6	36	6	Digit
7	37	7	Digit
8	38	8	Digit
9	39	9	Digit
A	41	12-1	Letter
B	42	12-2	Letter
C	43	12-3	Letter
D	44	12-4	Letter
E	45	12-5	Letter
F	46	12-6	Letter
G	47	12-7	Letter
H	48	12-8	Letter
I	49	12-9	Letter
J	4A	11-1	Letter
K	4B	11-2	Letter
L	4C	11-3	Letter
M	4D	11-4	Letter
N	4E	11-5	Letter
O	4F	11-6	Letter
P	50	11-7	Letter
Q	51	11-8	Letter
R	52	11-9	Letter
S	53	0-2	Letter

<u>ASCII Character</u>	<u>ASCII Code</u>	<u>Hollerith Punch (026)</u>	<u>Description</u>
T	54	0-3	Letter
U	55	0-4	Letter
V	56	0-5	Letter
W	57	0-6	Letter
X	58	0-7	Letter
Y	59	0-8	Letter
Z	5A	0-9	Letter
(space)	20	No punch	Blank
!	21	11-8-2	Exclamation point
"	22	8-7	Quotes
#	23	12-8-7	Number
\$	24	11-8-3	Dollar
%	25	0-8-5	Percent
&	26	8-2	Ampersand
'	27	8-4	Apostrophe
(28	0-8-4	Left parenthesis
)	29	12-8-4	Right parenthesis
*	2A	11-8-4	Asterisk
+	2B	12	Plus
,	2C	0-8-3	Comma
-	2D	11	Minus
.	2E	12-8-3	Period
/	2F	0-1	Slash
:	3A	8-5	Colon
;	3B	11-8-6	Semicolon
<	3C	12-8-6	Less than
=	3D	8-3	Equal
>	3E	8-6	Greater than
?	3F	12-8-2	Question
@	40	0-8-7	At
[5B	12-8-5	Left bracket
\	5C	0-8-2	Reverse slash
]	5D	11-8-5	Right bracket
^	5E	11-8-7	Circumflex
_	5F	0-8-6	Underline

FORTRAN COMPILATION ERRORS

L

Compilation errors are listed at the end of the source listing and indicated within the source listing in the following format:

Message

Significance

* $\begin{Bmatrix} N \\ F \end{Bmatrix}$, code, no., part

A compilation free of diagnostics is syntactically correct. The compilation is also free of common semantic errors, such as undefined variables in context requiring definition. If the detected error prevents code from being generated in a reasonably accurate manner, the error is considered fatal and compilation terminates. When an assumption is made as to the intended meaning of a statement, the diagnostic indicates the assumption. When possible, errors which may not be fatal (e.g., an A in column 3) are flagged. A reference to such a label (or the intended nonexistent label) would cause the fatal error.

N Trivial error; only flagged. Example: not separating array declarators in a dimension statement

F Fatal error

code Diagnostic number; see the following message for listing of codes

no. Number of statements in error; appears only when applicable

part Part of statement in error; appears only when applicable

variable

* $\begin{Bmatrix} N \\ F \end{Bmatrix}$, code

Compilation error. When errors cannot be detected until all the specification statements have been read and initially processed, the error appears in this format. As the specification statements are processed further, a few diagnostics can be printed. In these cases, the variable causing the difficulty is printed. The diagnostic is printed on the next line without a statement number reference since it is no longer available.

N Trivial error; only flagged. Example: not separating array declarators in a dimension statement

F Fatal error

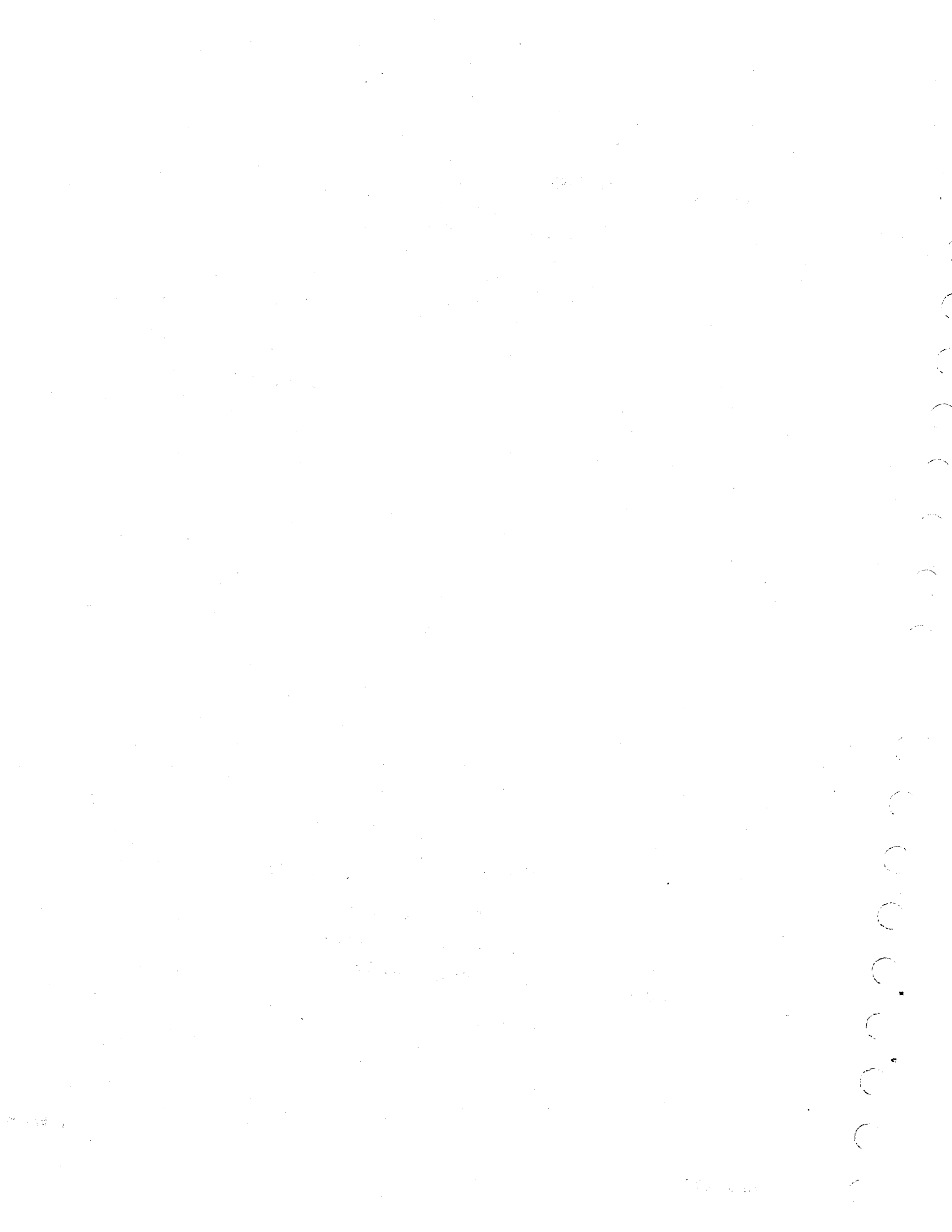
code Number of statements in error; appears only when applicable

<u>Message</u>	<u>Significance</u>
1	Field is not recognizable (illegal characters in field, such as 8 in octal field).
2	Minimum range limit of a constant is exceeded.
3	More than six characters in a name
4	Maximum range limit of a constant is exceeded.
5	Exponent is missing in a constant.
6	Subscripted variable was not previously dimensioned.
7	Expression in an IF statement does not have initial parenthesis.
8	Incorrect FORMAT statement
9	Illegal use of the .NOT. operator
10	Illegal operator or operand
11	Subprogram reference is illegal.
12	Labeled END card is illegal.
13	Number of arguments differs in references to the same subprogram.
14	Implied DO in DATA statement either contains wrong number of subscripts or subscript is out of range.
15	Expression has an illegal termination.
16	Unmatched parentheses in an expression
17	Relational operator is missing.
18	Relational operator used illegally.
19	Asterisk is assumed.
20	Only one ** is allowed per parentheses level.
21	A variable and a subprogram name are interchanged.
22	Subprogram name does not appear in an EXTERNAL statement.
23	One or more DO loops terminate on an undefined statement label.
24	Illegal subscript
25	Statement is syntactically incorrect.
26	This array was previously dimensioned in DIMENSION, COMMON, or TYPE statement or previously defined in an EXTERNAL statement. The previous dimensioning or defining is retained and the new ignored.
27	The field must be a variable or array name if processing a COMMON, DATA, EQUIVALENCE, BYTE, or SIGNED BYTE statement; an array name if processing a DIMENSION statement; or an array, variable, or FUNCTION name if processing a type statement.

<u>Message</u>	<u>Significance</u>
28	Logical IF statement contains another logical IF, DO, DATA, or FORMAT statement.
29	Name must be the name of an array.
30	Must be first statement of program unit.
32	A missing comma in this statement is assumed.
34	Illegal character in this statement is changed to a blank.
35	This line, which begins a statement, has other than zero or blank in column 6; blank is assumed.
36	Too many labeled common blocks declared, continuation of the last declared block is assumed.
37	The name in this COMMON statement is either a formal argument or defined in a previous COMMON statement. The name is ignored.
38	Name specified as two different types. This specification is ignored.
39	This byte typed as other than an integer, or it is a formal argument. The byte specification is ignored.
40	This byte previously specified as a different byte. The previous specification is retained and this specification is ignored.
41	The bit specified is not within bounds of the word size.
42	Least significant bit in this specification is greater than the most significant bit.
43	Name must be an external function or subroutine name.
44	Field must be a nonzero positive integer constant.
45	Array has more than three dimensions.
46	DATA statement contains too many constants for the space provided.
47	Statement has more than five continuation cards; excess cards are ignored.
48	An insufficient number of constants is provided in this data statement.
50	Constant is not same type as corresponding data cell.
51	Statement redefines DO loop parameter.
52	Statement type is unrecognizable; or it follows an executable statement.
53	Not defined
54	Statement label is meaningless; label is ignored.
55	Statement label previously defined; current label is ignored.

<u>Message</u>	<u>Significance</u>
56	Program name expected in this field.
57	Too many dimensions caused table overflow.
58	Symbol table overflowed; compilation terminates.
59	Statement label may not be zero.
60	No apparent exit from this program
61	Unclosed DO-implied list
62	Unformatted WRITE must have a list.
63	Name must be an integer variable or integer constant.
64	Name not implicitly an integer variable
65	A RETURN statement may appear only in a subroutine or function definition. A STOP statement is assumed.
66	Superflous information in this statement is ignored.
67	This field on STOP card must have an octal number not greater than 77777. STOP is assumed.
68	Field must be a positive integer.
69	Field must be an integer variable.
70	Field must be a statement label.
71	This form of ASSEM argument cannot reference elements in COMMON, EXTERNAL names, or subprogram arguments.
72	This type of statement may not terminate a DO loop.
73	This statement terminates a DO loop which is not the last DO encountered.
74	This GO TO jumps to itself.
75	A program consisting of only an END card is illegal.
77	Too many unique dummy parameter references caused a compiler table overflow.
78	Label in a DO statement must reference a statement following it.
79	Maximum allowable number of nested DOs exceeded. The DO loop may be implied in a DO list.
80	Subroutine argument table overflow; caused by large number of declared parameters and unique references to these parameters.
81	This formal argument was previously specified as another formal argument or the subprogram name.
82	Too many formal arguments caused a compiler table overflow.
83	The above name is not a variable or an array element.

<u>Message</u>	<u>Significance</u>
84	Two elements of the same array or common block are assigned to the same storage unit.
85	Blank common and formal arguments may not be initialized with DATA statements.
87	An array element in a BYTE, SIGNED BYTE, DATA, or EQUIVALENCE statement either has wrong number of subscripts or subscript is out of range.
88	Too many EQUIVALENCE names caused a compiler table overflow.
89	At least two elements must appear in an EQUIVALENCE statement.
90	The preceding equivalenced symbols have overflowed the origin of common.
91	DATA statement field is not an integer, real, double precision, or literal constant.
92	Missing terminating asterisk or quote in a literal string as appropriate
100	Catastrophic table overflow; compilation is abandoned. If the offending statement is arithmetic or a logical IF, the statement should be broken into two or more statements and the program recompiled.
101	Two PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statements in one program unit; the second is ignored.
103	Relative address argument in ASSEM statement requires an asterisk at the end of the preceding instruction.
110	Overflow of the table used for symbol references; subsequent references are not listed by option S processor.
111	The index used in this subscripted variable is in conflict with the dimension declaration.
113	Maximum number of macros overflowed; this macro definition is ignored.
114	This macro was previously defined; the new definition is ignored.
115	Call to an undefined macro.
116	Embedded macros are illegal.
152	Arithmetic table overflow



FORTRAN EXECUTION ERRORS

M

The following error messages apply to the FORTRAN I/O run-time only.

<u>Message</u>	<u>Significance</u>	<u>Action/Result</u>
1 I/O RQST statement no. ffff	Error in a format statement; illegal character in format statement ffff The current decimal value of the for- mat statement pointer	Program terminates
2 I/O RQST statement no. ffff gggg	Illegal character in the input field. ffff Current decimal value of format statement pointer gggg Current decimal value of input field pointer	Program terminates
3 I/O RQST statement no. ffff gggg	Input data exceeds limits of 16-bit word: Exponent $\left. \begin{array}{l} >39 \\ <-10 \end{array} \right\}$ ffff Current decimal value of format statement pointer gggg Current decimal value of the input field pointer	Program terminates
4 I/O RQST statement no. xx	Attempt to read on a write unit or write on a read unit xx Decimal unit number of a device used improperly	Program terminates
5 I/O RQST statement no. xx	Read or write request after an end-of-file has been read without first doing an EOF check xx Decimal unit number of a device used improperly	Program terminates

<u>Message</u>	<u>Significance</u>	<u>Action/Result</u>
7 I/O RQST statement no. xx	Write attempted on magnetic tape with no write enable xx The decimal unit number of a device used improperly	To continue press RETURN
8 I/O RQST statement no. xx	Attempt to use logical unit number greater than 99 xx The decimal unit number of a device used improperly	Program terminates
9 I/O RQST statement no. xx	Backspace at loadpoint xx The decimal unit number of a device used improperly	Program terminates
10 I/O RQST statement no. xx	End of magnetic tape sensed xx The decimal unit number of a device used improperly	To continue, press RETURN
12 I/O RQST statement no. ffff	Illegal formatted input; more elements are given than are contained in an input record ffff Current decimal value of format statement pointer	Program terminates
13 I/O RQST statement no. ffff	Illegal list; a list is given but there are no conversion codes in the format statement ffff Current decimal value of format statement pointer	Program terminates
14 I/O RQST statement no. xx	File defined twice; more than one OPEN request given for the same file xx Decimal file number for a mass storage device	Program terminates
15 I/O RQST statement no. xx	Parameter negative or zero; one of the parameters in an OPEN statement is negative or zero xx Decimal file number for a mass storage device	Program terminates

Message

Significance

Action/Result

16
I/O RQST
statement no.
xx

Sector address too large; the starting
sector address or ending address exceeds
 $2^{15}-1$

Program terminates

xx Decimal file number for a mass
storage device

17
I/O RQST
statement no.
xx

File not defined; a READ or WRITE request
was given for a file which was not defined
by an OPEN statement

Program terminates

xx Decimal file number for a mass
storage device

18
I/O RQST
statement no.
xx

Logical unit not a mass storage device

Program terminates

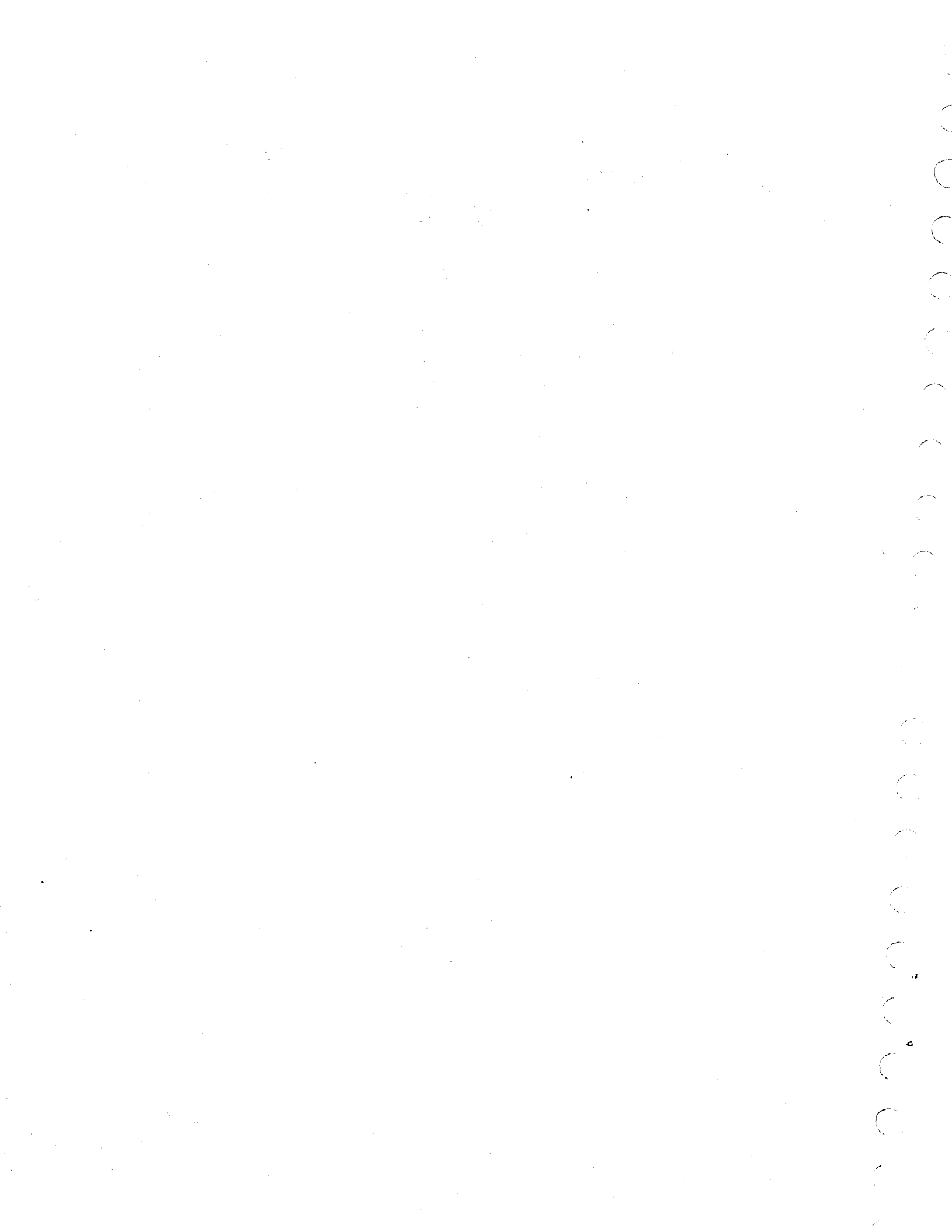
xx Decimal file number for a mass
storage device

19
I/O RQST
statement no.
xx

Record number in READ or WRITE request
incorrect. Resulting sector address is out
of the range of the file or it is zero

Program terminates

xx Decimal file number for a mass
storage device



FORTRAN MISCELLANEOUS ERRORS

N

<u>Message</u>	<u>Significance</u>
CORE OVFL	More than 32,767 cells of object code have been produced
*UD	Undefined symbol in address field
UNDEFINED SYMS name name name	Undefined statement labels and variable names
*SO	Scratch mass memory overflow
INPUT ERROR	Request from comment device for input has returned on error. FORTRAN will exit the job processor.

MS FORTRAN RESERVED WORD LIST

0

The following symbols may not be redefined in user programs.

DABS	Q8DFLT
DATAN	Q8DFNF
DBLE	Q8PKUP
DCOS	Q8PREP
DEXP	Q8PSE
DFIX	Q8PSEN
DFLOT	Q8QBCK
DFLT	Q8QD2D
DLOG	Q8QD2F
DSIGN	Q8QD2I
DSIN	Q8QEND
DSQRT	Q8QF2I
DSTOR1	Q8QF2I
DSTOR2	Q8QFLE
	Q8QFIX
FLOAT	Q8QFLT
FLOT	Q8QI2F
HFLOT	Q8QINI
HDFLOT	Q8QWND
	Q8QX
	Q8QX1
	Q8QX2
	Q8QX3
	Q8QY
	Q8QZ
	Q8STP
	Q8STPN
	RSTOR1

FORTRAN OPERATION IN ITOS USER AREA

P

At this time, FORTRAN does not fully support operation in the ITOS user area. Users who wish to use FORTRAN in this manner should be aware of the following restrictions:

- Blank common may not be used.
- Labeled common may be used only if the first program loaded does not declare labeled common. One method of accomplishing this is to make the main program a subroutine and to precede it with a program such as the following:

```
PROGRAM XYZ  
CALL MAIN  
CALL PGMOUT  
END
```

- The OPEN statement and the following forms of the READ/WRITE statement described in section 5 are not supported:

```
READ (k(n), f) list  
WRITE (k(n), f) list  
READ (k(n)) list  
WRITE (k(n)) list
```


INDEX

- Absolute addresses A-1
- Addresses
 - Absolute A-1
 - Relative A-1
- AFORM 9-22
- Allocated core, release of 9-30
- Alphanumeric conversion 6-20
 - Aw input/output 6-21
 - Rw input/output 6-21
- Arguments
 - Actual 7-3
 - Dummy 7-3
- Arithmetic assignment statements 5-1
- Arithmetic expressions 3-1
 - Mixed mode 3-3
 - Order of evaluation 3-2
 - Rules for forming 3-1
- Arithmetic IF 5-7
- Arrays G-1
 - Format specification in 6-27
- ASCII 9-21
 - Codes H-1
- ASSEM 5-34
- Assigned GO TO 5-4
- Assignment statements 5-1
 - Arithmetic 5-1
 - Label 5-2
- Asterisk or quote I/O 9-11
- Auxiliary I/O statements 5-28
 - BACKSPACE 5-29
 - ENDFILE 5-29
 - REWIND 5-28
- Aw input/output 6-20; 9-9

- BACKSPACE 5-29
- Basic external function 7-7
- Blank field specification 6-24
 - nX input 6-24
 - nX output 6-25
- BLOCK DATA 7-17
- Block data subprogram 7-17
- Blocks, tape 5-30

- Buffered input/output 9-31
- Byte statements 6-7
- Byte type data 2-5

- CALL 5-8; 7-16
- Calling sequences 9-13, 28, 29; A-1
- Character set K-1
- Character specifications, special 9-11
- Coding hints E-1
- Commands via the A/Q channel 9-30
- Comment line 4-1
- COMMON 6-2; P-1
- Common block 6-3
- Compilation 8-1
 - Errors L-1
 - Examples 8-3
 - Options 8-1
 - A 8-1
 - C 8-2
 - K 8-2
 - L 8-1
 - M 8-1
 - O 8-2
 - P 8-1
 - R 8-1
 - V 8-2
 - X 8-2
- Computed GO TO 5-6
- Constant
 - Double precision 2-7
 - Integer 2-3
 - Real 2-6
- CONTINUE 5-8
- Control statements 5-3
- Conversion
 - Alphanumeric 6-20
 - Double-precision 6-19
 - Format 9-5
 - Integer 6-13
 - Numeric 6-13
 - Real 6-15

Specifications 9-5
Dw.d input 6-19
Dw.d output 9-5
Ew.d input 9-7
Ew.d output 9-6
Iw input 9-8
Iw, Iw.d output 9-7
Rw input/output 9-10
Zw input 9-9
Zw output 9-8
\$w input 9-9
\$w output 9-8
Communication between FORTRAN
and assembly language programs A-1
Completion location 9-26
Core clock 9-30

DATA 6-9
Literals in 6-10
Data elements 2-1
Data format types
Byte 2-5
Double precision 2-7
Integer 2-3
Real 2-6
Signed byte 2-5
Single 2-5
Data types 2-4
DECHEX 9-21
Decimal character set K-1
DECODE 9-3, 17
Decremental DO loop 5-12
Descriptors, field 6-12
DFLOT D-1
DIMENSION 6-1
Dispatcher 9-30
DO 5-9
Extended range 5-16
Parameters 5-11
DO loop 5-9
Decremental 5-12
Nested 5-13
Structure 5-10
Transfer 5-15
Double precision
Constant 2-7
Conversion 6-19

Dw.d input 6-19
Dw.d output 6-20; 9-5
Floating point package D-1
Subscripted variable 2-8
Type data 2-7
Variable 2-8
Dw.d input 6-19
Dw.d output 6-20; 9-5

Editing specifications 6-21; 9-10
Literal free-field input/output 6-23
New line 9-11
nH input 6-21
nH output 6-22
Quote or asterisk I/O 9-11
wH, wX input/output 9-10
ENCODE/DECODE 9-3, 15
Error detection 9-19
Macros 9-17, 19
END 5-9
ENDFILE 5-29
EOF 7-8
EQUIVALENCE 6-4
Errors 9-15
Compilation L-1
Execution M-1
Format 9-15
Input/output 9-15
Miscellaneous N-1
Ew.d input 6-16
Ew.d output 6-18; 9-6
Executable statements 4-1; 5-1
ASSEM 5-34
Assignment 5-1
Arithmetic 5-1
Label 5-2
Control 5-3
Input/output 5-18
Execution 8-12
Errors M-1
Expressions 3-1
Arithmetic 3-1
Logical 3-4
Relational 3-3
Extended range of a DO 5-16
EXTERNAL 7-13