MOTOROLA
microsystems

M68FTN(D)

M6800
RESIDENT FORTRAN COMPILER
REFERENCE MANUAL

M68FTN(D)
March, 1977

M6800

RESIDENT FORTRAN COMPILER
REFERENCE MANUAL

First Edition
SEPTEMBER 1976

This manual is a reference manual and a prior knowledge of FORTRAN is required in most cases to use it. In no way is it intended to be a primer or tutorial.

Second Edition

TABLE OF CONTENTS

TABLE OF CONTENTS
(continued)

TABLE OF CONTENTS
(continued)

CHAPTER 1

INTRODUCTION

The Motorola M6800 EXORciser based FORTRAN compiler is a compact computer system designed for the solution of small to medium scale scientific and business problems. The system consists of the computer hardware plus software. The minimum configuration is:

- EXORciser
- 16,384 bytes of memory
- One EXORdisk drive unit (262,244 bytes)
- TTY or RS-232C compatible terminal
- M6800 Resident Editor
- M6800 Linking Loader

The minimum configuration may be expanded to include up to 65,536 bytes of memory and up to four disk drives for a total of over a million bytes of disk storage capability. In addition, a variety of hard copy and CRT terminals may be used.

The FORTRAN language is especially useful in writing programs for applications that involve mathematical computations and other manipulation of numerical data. The name FORTRAN is derived from FORmula TRANslator.

Source programs written in the FORTRAN language consist of a set of statements constructed by the programmer from the language elements described in this publication.

In a process called compilation, the program called the FORTRAN compiler analyzes the source program statements and translates them into a machine language program called the object program. After linking the object program with the M6800 Linking Loader, this program can be executed on the Motorola MC6800 Microprocessor. In addition, when the FORTRAN compiler detects errors in the source, it produces the appropriate diagnostic error messages. Note that it is possible to link FORTRAN and assembly language programs together.

1.1 The Resident FORTRAN Compiler is invoked with the EDOS-II FORT command. This command and its parameters are defined as follows.

FORMAT:

```
FORT,list,object,source-1,source-2,----source-n
```

PURPOSE:

To compile FORTRAN source file(s) with the M6800 FORTRAN compiler and to produce a listing (optional) and relocatable object file (optional).

COMMENTS:

list -- Listing file or device (allowable devices: #LP-line printer, #CN-console)
If a listing file is selected, then an object file cannot be specified. If entry is null, no listing will be produced.
#LP8 or #CN8 causes conditional compilation records to be compiled. See section 2.2.
#LP3 or #CN3 will prohibit paging of the source listing.

object -- Relocatable object file
An object file cannot be specified if a listing file is specified. If entry is null, no object will be produced.

source -i -- File(s) containing the source program(s)
More than one program or subroutine may be specified as source input. A source file may contain a portion of a program or several programs.

EXAMPLE:

```
FORT,#LP,OBJ,PROG1
```

Produces a listing including source, and symbol table on the line printer and a relocatable object file, "OBJ", for the source program(s) on file "PROG1".

1.2 Line and Character Delete

During execution of a source program an incorrectly entered line of data at the ? may be deleted by holding down the CTRL (control) key then pressing the X key. This will delete the data, give a carriage return and a line feed, at which you may re-enter input. A character can be deleted by holding down the CTRL (control) key and pressing the DEL or rubout key. One character is deleted each time the DEL or rubout key is pressed.

1.3 The Resident FORTRAN Compiler is invoked with the MDOS FORT command. This command is defined briefly as follows.

FORMAT:

FORT,SOURCE-1,SOURCE-2,---,SOURCE-n

PURPOSE:

To compile source program statements written in M6800 FORTRAN language and to produce a listing (optional) and relocatable object file (optional).

COMMENTS:

source-i -- File(s) containing the source program(s). More than one program or subroutine may be specified as source input. A source file may contain a portion of a program or several programs. The relocatable object file will have the same name as source-i with a default suffix of "RO". If there are more than one source file, the output will be strung together.

For a full description of the MDOS FORT command, see the MDOS command manual.

EXAMPLE:

FORT PROG1

This produces a listing including source on the line printer and a relocatable object file PROG1.RO for the source program(s) on file PROG1.

CHAPTER 2

FORTRAN COMPILER

2.1 STATEMENTS

Source programs consist of a set of statements from which the compiler generates machine instructions, constants, and storage areas. A given FORTRAN statement effectively performs one of three functions:

1. Causes certain operations to be performed (e.g., addition, multiplication, branching).

2. Specifies the nature of the data being handled.

3. Specifies the characteristics of the source program.

FORTRAN statements usually are composed of certain FORTRAN key words used in conjunction with the basic elements of the language; constants, variables, and expressions. The categories of FORTRAN statements are as follows:

1. Arithmetic Assignment Statements: These statements cause calculations to be performed or conditions to be tested. The result replaces the current value of a designated variable or subscripted variable.

2. Control Statements: These statements enable the user to govern the flow of and to terminate the execution of the object program.

3. Input/Output Statements: These statements enable the user to transfer data between internal storage and the terminal or disk.

4. FORMAT Statement: This statement is used in conjunction with input/output statements to specify the form of a FORTRAN record.

5. DATA Initialization Statement: This statement is used to assign initial values to variables.

6. Specification Statements: These statements are used to declare the properties of variables and arrays.

7. Subprogram Statements: These statements enable the user to name and define functions and subroutines, which can be compiled following the main program as one source file or as a separate file not existing with the main program.

8. OPENF/CLOSEF Statements: These statements are used in conjunction with the disk I/O to assign a FORTRAN file number to a file name and set up the EDOS pointers.

2.2 CODING FORTRAN STATEMENT

The statements of a FORTRAN source program can be entered on a terminal with each line representing one 72-column line. If a statement is too long for one line, it may be continued on successive lines by placing an & symbol in column 1 of each continuation line.

As many blanks as desired may be written between keywords and variable names to improve readability. Each keyword must have at least one blank following it. Blanks that are inserted in literal data are retained and treated as blanks within the data. Variable names, keywords, and numbers may not contain embedded blanks.

Comments to explain the program may be written in columns 2 through 72 of a line if the letter C is placed in column 1. Comments may appear between FORTRAN statements; a comment line may not immediately precede a continuation line. Comments are ignored by the FORTRAN compiler. Blanks may be inserted where desired to improve readability.

The C indicating a comment record, the character & signifying statement continuation, and an X for conditional compilation must start in column one. If an X is in column one, the record is treated as a comment unless #LP8 or #CN8 is used when invoking the compiler. In this case, records with an X in column one will be compiled. Statement numbers ranging from 1-99999 also start in column one and are followed by at least one blank. All other statements may start anywhere from 2-72.

2.3 ELEMENTS OF THE LANGUAGE

The basic elements of the language are discussed in the following paragraphs. The actual FORTRAN statements in which these elements are used are discussed in subsequent chapters. The term program unit refers to a main program or a subprogram.

The order of a FORTRAN program unit is as follows:

1. Subprogram statement, if any.
2. COMMON and DIMENSION statements, if any. They may be intermixed.
3. DATA statements, if any.
4. Executable statements, at least one of which must be present.
5. END statement.

FORMAT and DATA statements may appear anywhere before the END statement. DATA statements, however, must follow any specification statements that contain the same variable or array names.

2.4 CONSTANTS

A constant is a fixed, unvarying quantity. There are two classes of constants -- those that specify numbers (numerical constants), and those that specify literal data (literal constants).

Numerical constants may be integer or real numbers; and literal constants may be a string of alphameric and/or special characters.

2.4.1 Integer Constants

DEFINITION An Integer Constant is a whole number written without a decimal point. It occupies two bytes of memory. The maximum magnitude is 32,767 ($2^{15}-1$).

An integer constant may be positive, zero, or negative; if unsigned, it is assumed to be positive. Its magnitude must not be greater than the maximum and it must not contain embedded commas.

EXAMPLES

Valid Integer Constants

```
0
91
173
-21474
```

Invalid Integer Constants

```
27.          (contains a decimal point)
3145903612   (exceeds the allowable range)
5,396        (contains an embedded comma)
```

2.4.2 Real Constants

DEFINITION A Real Constant has one of three forms: a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal point. A real constant occupies four bytes of memory.

A basic real constant is a string of up to eight decimal digits with a decimal point. The basic real constant occupies four storage locations (bytes).

The magnitude of the real constant is 0 or 16^{-65} (approximately 10^{-78}) through 16^{63} (approximately 10^{75}).

The precision using four bytes is six hexadecimal digits (approximately 7.2 decimal digits).

A real constant may be positive, zero, or negative (if unsigned, it is assumed to be positive) and must be of the allowable magnitude. It may not contain embedded commas. The decimal exponents permit the expression of a real constant as the product of a basic real constant or integer constant times 10 raised to a desired power.

EXAMPLES

Valid Real Constants

```
+0
-999.9999
7.0E 0          (i.e., 7.0 x 10^0 = 7.0)
19761.25E 1     (i.e., 19761.25 x 10 = 197612.5)
7.E3
7.0E3           (i.e., 7.0 x 10^3 = 7000.0)

7E-03           (i.e., 7.0 x 10^-3 = 0.007)
```

Invalid Real Constants

```
1               (Missing a decimal point or a decimal exponent)
3,471.1         (Embedded comma)
1.E             (Missing a 1- or 2-digit integer constant
                 following the E.  Note that it is not interpreted
                 as 1.0 x 10^0.)
1.2E 113        (E is followed by a 3-digit integer constant)
23.5E 97        (Magnitude outside the allowable range; that
                 is 23.5 x 10^97 >16^63)
21.3E-90        (Magnitude outside the allowable range; that
                 is 21.3 x 10^-90 <16^-65)
```

2.4.3 Literal Constants

DEFINITION A Literal Constant is a string of alphanumeric and/or special characters, enclosed in apostrophes.

The string may contain any characters. Each character requires one byte of storage. The number of characters in the string, including blanks, may not be greater than 72.

Literals may be used in FORMAT, DATA, and assignment statements. Literals also may be used as the actual arguments in a CALL statement. However, literals used in DATA and assignment statements and as arguments in a CALL statement are limited to two bytes and may be used only with integers.

EXAMPLES

```
'DATA'
'X-COORDINATE     Y-COORDINATE     Z-COORDINATE'
'3.14'
K = 'AB'
```

2.5 SYMBOLIC NAMES

DEFINITION Symbolic Names are from 1 through 6 alphanumeric characters (i.e., numerics 0 through 9 and alphabetic A through Z), the first of which must be alphabetic. No element, such as GOTO, IF, FORMAT, etc... may be used as a symbolic name. All elements are considered reserved words.

DESCRIPTION Symbolic Names are used in a program unit (i.e., a main program or a subprogram) to identify elements in the following classes.

- An array and the elements of that array (see "ARRAYS")
- A variable (see "VARIABLES")
- An intrinsic function
- A FUNCTION subprogram (see "FUNCTION Subprograms")
- A SUBROUTINE subprogram (see "SUBROUTINE Subprograms")

Symbolic names must be unique within a class in a program unit and can identify elements of only one class with the following exception.

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

Once a symbolic name, or an external procedure name is used in any unit of an executable program, no other program unit of that executable program can use that name to identify an entity of these classes in any other way.

2.6 VARIABLES

A FORTRAN variable is a symbolic representation of a quantity that occupies a storage area. The value specified by the name is always the current value stored in the area.

For example, in the statement:

```
A = 5.0+B
```

both A and B are variables. The value of B is determined by some previous statement and may change from time to time. The value of A is calculated whenever this statement is executed and changes as the value of B changes.

2.6.1 Variable Name

Using meaningful variable names can serve as an aid in documenting a program. That is, someone other than the programmer may look at the program and understand its function. For example, the equation to compute the distance a car travels in a given period of time at a given rate of speed could be written.

```
X = Y * Z
```

Where * designates multiplication. However, it would be more meaningful to an individual reading this equation if the programmer had written

```
DIST = RATE * TIME
```

EXAMPLES

Valid Variable Names:

B292S

RATE

VAR

Invalid Variable Names:

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

2.6.2 Variable Types and Lengths

The type of a variable corresponds to the type of data the variable represents. Thus, an integer variable represents integer data or literal data and a real variable represents real data.

For each type of variable, there is a corresponding number of storage locations (bytes) that are reserved for the variable. The following list shows each variable type with its associated length:

Variable Type	Length
Integer	2
Real	4

2.6.3 Type Declaration By The Predefined Specification

The predefined specification is a convention used to specify variables as integers or reals as follows:

1. If the first character of the variable name is I, J, K, L, M, or N, the variable is integer of length 2.

2. If the first character of the variable name is any other alphabetic character, the variable is real of length 4.

This convention is the traditional FORTRAN method of implicitly specifying the type of a variable as being either integer or real. In all examples that follow in this publication it is presumed that this specification applies.

2.7 ARRAYS

DESCRIPTION A FORTRAN array is a set of variables identified by a single variable name. A particular variable in the array may be referred to by its position in the array (e.g., first variable, third variable, seventh variable, etc.). Consider the array named NEXT which consists of five variables, each currently representing the following values: 273, 41, 8976, 59, and 2.

NEXT(1) is the location containing 273

NEXT(2) is the location containing 41

NEXT(3) is the location containing 8976

NEXT(4) is the location containing 59

NEXT(5) is the location containing 2

Each variable (element) in this array consists of the name of the array (i.e., NEXT) immediately followed by a number enclosed in parentheses, called a subscript quantity. The variables which the array comprises are called subscripted variables. Therefore, the subscripted variable NEXT(1) has the value 273; the subscripted variable NEXT(2) has the value 41, etc.

The subscripted variable NEXT(I) refers to the "Ith" subscripted variable in the array, where I is an integer variable that may assume a value of 1, 2, 3, 4, or 5.

To refer to any element in an array, the array name must be subscripted. In particular, array name alone does not represent the first element.

Consider the following array named LIST described by two subscript quantities, the first ranging from 1 through 5, the second from 1 through 3:

	Column 1	Column 2	Column 3
ROW 1	82	4	7
ROW 2	12	13	14
ROW 3	91	1	31
ROW 4	24	16	10
ROW 5	2	8	2

Suppose it is desired to refer to the number in row 2, column 3; this would be:

LIST (2,3)

Thus, LIST (2,3) has the value 14 and LIST (4,1) has the value 24.

Ordinary mathematical notation might use LIST to represent any element of the array LIST. In FORTRAN, this is written as LIST(I,J) where I equals 1, 2, 3, 4, or 5 and J equals 1, 2, or 3.

2.7.1 Declaring The Size And Type Of An Array

The size (number of elements) of an array is specified by the number of subscript quantities of the array and the maximum value of each subscript quantity. This information must be given for all arrays before using them in a FORTRAN program so that an appropriate amount of storage may be reserved. Declaration of this information is made by a DIMENSION statement or a COMMON statement. These statements are discussed in detail in Chapter 7 "SPECIFICATION STATEMENTS." The type of an array name is determined by the conventions for specifying the type of a variable name. Each element of an array is of the type specified for the array name.

2.7.2 Arrangement Of Arrays In Storage

Arrays are stored in ascending storage locations, with the value of the first of the subscript quantities increasing most rapidly and the value of the last increasing least rapidly.

For example, the array LIST, whose values are given in the previous example, is arranged in storage as follows:

82 12 91 24 2 4 13 1 16 8 7 14 31 10 2

The array named A, described by one subscript quantity which varies from 1 to 5, appears in storage as follows:

A(1) A(2) A(3) A(4) A(5)

The array named B, described by two subscript quantities with the first subscript quantity varying over the range from 1 to 5, and the second varying from 1 to 3, appears in ascending storage locations in the following order:

B(1,1) B(2,1) B(3,1) B(4,1) B(5,1)

B(1,2) B(2,2) B(3,2) B(4,2) B(5,2)

B(1,3) B(2,3) B(3,3) B(4,3) B(5,3)

Note that B(1,2) and B(1,3) follow in storage B(5,1) and B(5,2), respectively.

The following list is the order of an array named C, described by three subscript quantities with the first varying from 1 to 3, the second varying from 1 to 2, and the third varying from 1 to 3:

C(1,1,1) C(2,1,1) C(3,1,1) C(1,2,1) C(2,2,1) C(3,2,1)

C(1,1,2) C(2,1,2) C(3,1,2) C(1,2,2) C(2,2,2) C(3,2,2)

C(1,1,3) C(2,1,3) C(3,1,3) C(1,2,3) C(2,2,3) C(3,2,3)

Note that C(1,1,2) and C(1,1,3) follow in storage C(3,2,1) and C(3,2,2), respectively.

2.8 SUBSCRIPTS

DESCRIPTION A subscript is an integer subscript quantity or a set of integer subscript quantities separated by commas, that is used to identify a particular element of an array. The number of subscript quantities in any subscript must be the same as the number of dimensions of the array with which the subscript is associated. A subscript is enclosed in parentheses and is written immediately after the array name. A maximum of three subscript quantities can appear in a subscript. Valid types are: integer constant, integer variable, or integer variable $\pm$ integer constant.

The following restrictions apply to the construction of subscript quantities.

1. Subscript quantities may not contain arithmetic expressions that use any of the arithmetic operators: *,/,**.

2. Subscript quantities may not contain function references.

3. Subscript quantities may not contain subscripted names.

4. Variable subscripts must be integer only (not real).

5. The evaluated result of a subscript quantity should always be greater than zero and less than or equal to the size of the corresponding dimension.

A subscript may have one of the following forms:

1. Positive integer constant i.e., 3, 21, 418

2. Integer variable

3. Integer variable plus/minus constant i.e., NOX+3, IX-5

EXAMPLES

Valid Subscripted Variables:

ARRAY (IHOLD)

NEXT (19)

MATRIX (I-5)

Invalid Subscripted Variables:

ARRAY (-5)	(A subscript quantity may not be negative)
LOT (0)	(A subscript quantity may never be nor assume a value of zero)
ALL(X)	(A subscript quantity may not be a real variable)

2.9 EXPRESSIONS

The value of an arithmetic expression is always a number whose type is integer or real.

2.9.1 Arithmetic Expressions

The simplest arithmetic expression consists of a primary which may be a single constant, variable, subscripted variable, function reference, or another expression enclosed in parentheses. The primary may be either integer or real.

In an expression consisting of a single primary, the type of the primary is the type of the expression.

EXAMPLES

Primary	Type of Primary	Type of Expression
3	Integer constant	Integer of length 2
A	Real variable	Real of length 4
3.14E3	Real constant	Real of length 4
SIN(X)	Real function reference	Real of length 4
(A*B+C)	Parenthesized real expression	Real of length 4

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

2.9.2 Arithmetic Operators

The arithmetic operators are as follows;

Arithmetic Operator	Definition
**	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction

2.9.3 Rules for Constructing Arithmetic Expressions

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

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

   AB

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

   A*B or B*A

2. No two arithmetic operators may appear in sequence in the same expression.

   For example, the following expressions are invalid:

   A*/B and A***B

   The expression A* -B is an exception and is treated as

   A*(-B)

   In effect, -B will be evaluated first and then A will be multiplied with it. (For further uses of parentheses, see rule 3.)

3. Order of Computation: Computation is performed from left to right according to the hierarchy of operations shown in the following list.

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

This hierarchy is used to determine which of two consecutive operations is performed first. If the first operator is higher than or equal to the second, the first operation is performed. If not, the second operator is compared to the third, etc. When the end of the expression is encountered, all of the remaining operations are performed in reverse order.

For example, in the expression A*B+C*D**I, the operations are performed in the following order:

a. A*B Call the result X (multiplication) (X+C*D**I)

b. D**I Call the result Y (exponentiation) (X+C*Y)

c. C*Y Call the result Z (multiplication) (X+Z)

d. X+Z Final operation (addition)

A unary minus has the highest hierarchy. Thus,

A= -B is treated as A=0-B

A= -B*C is treated as A=(-B)*C

A= -B+C is treated as A=(-B)+C

Parentheses may be used in arithmetic expressions, as in algebra, to specify the order in which the arithmetic operations are to be computed. Where parentheses are used, the expression within the parentheses is evaluated before the result is used. This is equivalent to the definition above since a parenthesized expression is a primary.

For example, the following expression:

B+((A+B)*C)+A**2

is effectively evaluated in the following order:

a.	(A+B)	Call the result X	B+(X*C)+A**2
b.	(X*C)	Call the result Y	B+Y+A**2
c.	B+Y	Call the result W	W+A**2
d.	A**2	Call the result Z	W+Z
e.	W+Z	Final operation	

4. The type and length of the result of an operation depends upon the type and length of the two operands (primaries) involved in the operation. Table 2-1 shows the type and length of the result of the operations +, -, *, and /.

TABLE 2-1. DETERMINING THE TYPE AND LENGTH OF THE RESULTS OF +, -, *, / OPERATIONS

+ - * /	INTEGER (2)	REAL (4)
INTEGER (2)	Integer (2)	Real (4)
REAL (4)	Real (4)	Real (4)

NOTE

When division is performed using two integers, the answer is truncated and an integer answer is given. For example, if I=9 and J=2, then the expression (I/J) would yield an integer answer of 4 after truncation.

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

Variable Names	Type	Length Specification
C, D	Real Variable	4, 4
I, J, K	Integer Variable	2, 2, 2

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

Subexpression	Type and Length
I*J (Call the result M)	Integer of length 2
C**K (Call the result Y)	Real of length 4
M/Y (Call the result Z)	Real of length 4
Z+D	Real of length 4

Thus, the final type of the entire expression is real of length 4, but the type changed at different stages in the evaluation. Note that, depending on the values of the variables involved, the result of the expression I*J*C might be different from I*C*J.

5. The arithmetic operator denoting exponentiation (i.e., **) may only be used to combine the types of operands shown in Table 2-2.

TABLE 2-2. VALID COMBINATIONS WITH THE ARITHMETIC OPERATOR **

Base		Exponent
Integer	**	Integer
Real	**	Integer

2.9.4 Logical Expressions

DESCRIPTION A logical expression consists of two arithmetic expressions, which may be simple variables, connected by one of the following relational operators:

.EQ. - equal

.NE. - not equal

.GT. - greater than

.LT. - less than

.GE. - greater than or equal to

.LE. - less than or equal to

EXAMPLES

```
C.EQ.C

C+5.0.NE.21

(C+D)*E.GT.50
```

It should be clearly understood here that arithmetic expressions involved in relational operations are evaluated first before the relational operation is applied.

Relational operations in turn may be connected by the use of the logical connectives .AND. and .OR.:

```
C.EQ.D.OR.E.EQ.F

C.NE.D.AND.E.GT.F.OR.G.EQ.H
```

Normally .AND. operations have a higher hierarchy than .OR. operations, thus C.EQ.D.AND.E.GT.F.OR.G.EQ.H is evaluated as

```
(C.EQ.D.AND.E.GT.F) .OR.G.EQ.H
```

However parentheses may be used to change the order or evaluation -

```
C.EQ.D.AND.(E.GT.F.OR.G.EQ.H)
```

The meaning of a logical operation may be reversed by the modifier .NOT..

```
.NOT.(W.EQ.Y.AND.Z.EQ.V)
```

means everything but the intersection of W.EQ.Y.AND.Z.EQ.V

CHAPTER 3

ARITHMETIC ASSIGNMENT STATEMENT

3.1 GENERAL FORM

The general form is

a = b

Where: a is a subscripted or nonsubscripted variable
b is an arithmetic expression

This FORTRAN statement closely resembles a conventional algebraic equation; however, the equal sign specifies replacement rather than equality. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable to the left of the equal sign.

Table 3-1 gives the conversion rules used for placing the evaluated result of arithmetic expression b into variable a.

3.2 ASSIGNMENT STATEMENTS

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

Variable Names	Type	Length Specification
I, J, K	Integer Variables	2
A, B, C, D	Real variables	4

Then the following examples illustrate valid arithmetic statements using constants, variables, and subscripted variables of different types:

Statements	Description
A = B	The value of A is replaced by the current value of B.
K = B	The value of B is truncated to an integer value, and this value replaces the value of K.
A = I	The value of I is converted to a real value, and this result replaces the value of A.
I = I + 1	The value of I is replaced by the value of I + 1.
A = C*D	The product of C and D replaces the value of A.

TABLE 3-1. CONVERSION RULES FOR THE ARITHMETIC ASSIGNMENT STATEMENT
a = b

Type of a \ Type of b	INTEGER	REAL
INTEGER	Assign	Fix and Assign
REAL	Float and assign	Assign

1. Assign means transmit the resulting value, without change. If the significant digits of the resulting value exceed the specified length, results are unpredictable.

2. Fix means transform the resulting value to the form of a real constant and truncate the fractional portion.

3. Float means transform the resulting value to the form of a REAL number, retaining in the process as much precision of the value as a REAL number can contain.

CHAPTER 4

CONTROL STATEMENTS

4.1 INTRODUCTION

Normally, FORTRAN statements are executed sequentially. That is, after one statement has been executed, the statement immediately following it is executed. This section discusses the statements that may be used to alter and control the normal sequence of execution of statements in the program.

4.2 GO TO STATEMENTS

GO TO statements permit transfer of control to an executable statement specified by number in the GO TO statement. Control may be transferred either unconditionally or conditionally. The GO TO statements are:

1. Unconditional GO TO statement
2. Computed GO TO statement

4.2.1 Unconditional GO TO Statement

GENERAL FORM

GO TO XXXX

Where: XXXX is an executable statement number.
The GO TO must be separated by a blank.

This GO TO statement causes control to be transferred to the statement specified by the statement number. Every subsequent execution of this GO TO statement results in a transfer to that same statement. Any executable statement immediately following this statement must have a statement number; otherwise it can never be referred to or executed.

EXAMPLE

```
      GO TO 25

   10 A = B + C

         .
         .
         .

   25 C = E**2

         .
         .
         .
```

In this example, each time the GO TO statement is executed, control is transferred to statement 25.

4.2.2 Computed GO TO Statement

GENERAL FORM

GO TO (x_1, x_2, x_3,...,x_n), i

Where: x_1, x_2,...,x_n, are executable statement numbers.

i is a nonsubscripted integer variable whose current value is in the range: $1 \leq i \leq n$.

The GO TO must be separated by a blank.

This statement causes control to be transferred to the statement numbered x_1, x_2, x_3,..., or x_n, depending on whether the current value of i is 1, 2, 3,..., or n, respectively. The index i is checked at execution time to ensure it is within the range $1 \leq i \leq n$. If the index is less than 1, execution will continue at statement x_1. If the index i is greater than n, execution will continue at statement x_n. No error message will be given.

EXAMPLE

```
      GO TO (25, 10, 7), ITEM
         .
         .
         .
    7 C = E**2+A
         .
         .
         .
   25 L = C
         .
         .
         .
   10 B + 21.3E02
```

In this example, if the value of the integer variable ITEM is 1, statement 25 will be executed next. If ITEM is equal to 2, statement 10 is executed next, and so on.

4.3 ARITHMETIC CONTROL STATEMENTS

4.3.1 Arithmetic IF Statement

GENERAL FORM

IF (a) x_1, x_2, x_3

Where: a is any arithmetic expression.
x_1, x_2, x_3 are any executable statement numbers.

The arithmetic IF statement causes control to be transferred to the statement numbered x_1, x_2, or x_3 when the value of the arithmetic expression (a) is less than, equal to, or greater than zero, respectively. The first executable statement following the arithmetic IF statement must have a statement number; otherwise, it can never be referred to or executed.

EXAMPLE

```
      IF (A(J,K)**3-B)10, 4, 30
                  .
                  .
                  .
    4 D = B + C
                  .
                  .
                  .
   30 C = D**2
                  .
                  .
                  .
   10 E = (F*B)/D+1
                  .
                  .
                  .
```

In this example, if the value of the expression (A(J,K)**3-B) is negative, the statement numbered 10 is executed next. If the value of the expression is zero, the statement number 4 is executed next. If the value of the expression is positive, the statement numbered 30 is executed next.

4.3.2 Logical IF Statement

GENERAL FORM

IF (a), s

Where: a is any logical expression.
s is any valid executable FORTRAN statement except IF or DO.

The statement s is executed if the expression a is true; otherwise, the next executable statement following the logical IF statement is executed. The statement following the logical IF will be executed in any case after the statement s, unless the statement s causes a transfer.

EXAMPLES

```
IF (FLAG1.OR.FLAG2) GO TO 123
IF (A*B.LT.1.23) CALL RATE
IF (.NOT.(A.LT.6.0.OR.B.GT.5.0) RETURN
```

If only a variable name is given as a, the variable will be considered true and the statement s will be executed if the named variable is positive (greater than or equal to zero). The variable will be considered false and the statement s will not be executed if the named variable is negative.

```
IF (MONDAY) GO TO 10
```

NOTE

If the expression (a) is real, a test for exact zero, or a test with the logical operator.EQ., may not be meaningful on a binary machine. If the expression involves any amount of computation, a very small value is more likely to result than a zero. When testing to branch equal using .EQ., the value may be very close and differ only in the least significant bit. For this reason, IF statements using real floating-point numbers should not be programmed to have a zero or .EQ. value.

4.3.3 DO Statement

GENERAL FORM

	End of Range	DO Variable	Initial Value	Test Value	Increment
DO	x	i =	m_1,	m_2,	m_3

Where: x is an executable statement number appearing after the DO statement.
i is a nonsubscripted integer value and cannot be a dummy. m_1, m_2, and m_3 are either unsigned integer constants greater than zero or unsigned nonsubscripted integer variables whose value is greater than zero. m_2 may not exceed $2^{16}-2$ in value. m_3 is optional; if it is omitted, its value is assumed to be 1. In this case, the preceding comma must also be omitted.
The DO and x must each be separated by a blank.
Values m1, m2, or m3 may not be an expression

The DO statement is a command to execute, at least once, the statements that physically follow the DO statement, up to and including the statement numbered x. These statements are called the range of the DO. The first time the statements in the range of the DO are executed, i is initialized

to the value m_1; each succeeding time i is increased by the value m_3. When, at the end of the iteration, i is equal to the highest value that does not exceed m_2, control passes to the statement following the statement numbered x. Thus, the number of times the statements in the range of the DO are executed is given by the expression:

$$\left[\frac{m_2 - m_1}{m_3}\right] + 1$$

where the brackets represent the largest integral value not exceeding the value of the expression within the brackets. If m_2 is less than m_1, the statements in the range of the DO are executed once. Upon completion of the DO, the DO variable is undefined and may not be used until assigned a value (e.g., in a READ list).

There are several ways in which looping (repetitively executing the same statements) may be accomplished when using the FORTRAN language. For example, assume that a manufacturer carries 1000 different machine parts in stock. Periodically, he may find it necessary to compute the amount of each different part presently available. This amount may be calculated by subtracting the number of each item used, OUT (I), from the previous stock on hand, STOCK (I).

EXAMPLE 1

```
      .
      .
      .
      I=0
10    I=I+1
      STOCK(I)=STOCK(I)-OUT(I)
      IF(I-1000) 10, 30, 30
30    A=B+C
      .
      .
      .
```

The first, second, and fourth statements required to control the previously shown loop could be replaced by a single DO statement, as shown in Example 2.

EXAMPLE 2

```
      .
      .
      .
      DO 25 I = 1, 1000
25    STOCK(I) = STOCK(I) - OUT(I)
      A = B+C
      .
      .
      .
```

In Example 2, the DO variable, I is set to the initial value of 1. Before the second execution of statement 25, I is increased by the increment, 1, and statement 25 is again executed. After 1000 executions of the DO loop, I equals 1000. Since I is now equal to the highest value that does not exceed the test value, 1000, control passes out of the DO loop and the third statement is executed next. Note that the DO variable I is now undefined; its value is not necessarily 1000 or 1001.

EXAMPLE 3

```
          .
          .
          .
      DO 25 I = 1, 10, 2
      J = I + K
   25 ARRAY(J) = BRAY(J)
      A = B + C
          .
          .
          .
```

In Example 3, statement 25 is the end of the range of the DO loop. The DO variable, I, is set to the initial value of 1. Before the second execution of the DO loop, I is increased by the increment, 2, and the second and third statements are executed a second time. After the fifth execution of the DO loop, I equals 9. Since I is now equal to the highest value that does not exceed the test value, 10, control passes out of the DO loop and the fourth statement is executed next. Note that the DO variable I is now undefined; its value is not necessarily 9 or 11.

4.3.4 Programming Considerations in Using a DO Loop

1. The indexing parameters of a DO statement (i, m_1, m_2, m_3) should not be changed by a statement within the range of the DO Loop.

2. There may be other DO statements within the range of DO statement. All statements in the range of an inner DO must be in the range of the outer DO. A set of DO statements satisfying this rule is called a nest of DO's.

EXAMPLE 1

```
      DO 50 I = 1,4 ----------------------------+
      A(I) = B(I)**2                            |
      DO 50 J = 1, 5 --+ Range of               | Range of
   50 C(J+1) = A(I) ---+ inner DO               | outer DO
                                                |
                                                +
```

EXAMPLE 2

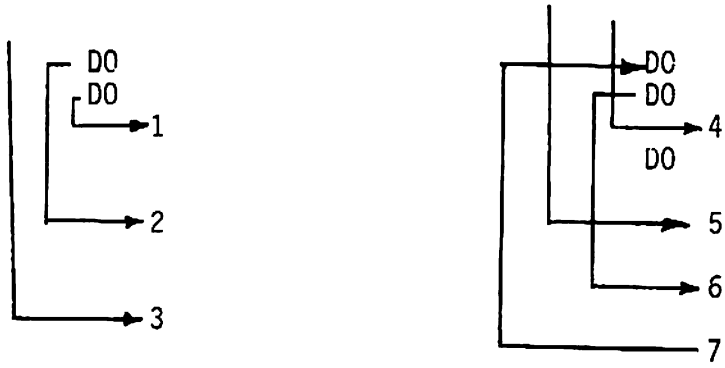
```
      DO 10 INDEX = L, M ──────────────────────┐
      N = INDEX + K                            │ Range of
      DO 15 J = 1, 100, 2 ┐ Range of           │ outer DO
   15 TABLE(J) = SUM(J,N)-1 ┘ inner DO         │
   10 B(N) = A(N) ──────────────────────────────┘
```

3. A transfer out of the range of any DO loop is permissible at any time.

4. The extended range of a DO is defined as those statements in the program unit containing the DO statement that are executed between the transfer out of the innermost DO of a nest of DO's and the transfer back into the range of this innermost DO. The following restrictions apply:
   - Transfer into the range of a DO is permitted only if such a transfer is from the extended range of the DO.
   - The extended range of a DO statement must not contain another DO statement that has an extended range if the second DO is within the same program unit as the first.
   - The indexing parameters (i, m_1, m_2, m_3) cannot be changed in the extended range of the DO.

Note that a statement that is the end of the range of more than one DO statement is within the innermost DO. The statement label of such a terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

EXAMPLE



In the preceding example, the transfers specified by the numbers 1, 2, and 3 are permissible, whereas those specified by 4, 5, 6, and 7 are not.

5. The indexing parameters (i, m_1, m_2, m_3) may be changed by statements outside the range of the DO statement only if no transfer is made back into the range of the DO statement that uses those parameters.

6. The last statement in the range of a DO loop (statement x) must be an executable statement. It cannot be a GO TO statement of any form, or a STOP, RETURN, arithmetic IF statement, or another DO statement.

7. The use of, and return from, a subprogram from within any DO loop in a nest of DO's is permitted.

4.3.5 CONTINUE Statement

GENERAL FORM

```
CONTINUE
```

CONTINUE is a dummy statement that may be placed anywhere in the source program without affecting the sequence of execution. It may be used as the last statement in the range of a DO in order to avoid ending the DO loop with a GO TO, STOP, RETURN, arithemetic IF, or another DO statement.

EXAMPLE 1

```
                .
                .
                .
  DO 30 I = 1, 20
7 IF (A(I)-B(I))  5, 30, 30
5 A(I) = A(I) + 1.0
  B(I) = B(I) - 2.0
                .
                .
                .
  GO TO 7
30 CONTINUE
  C = A(3) + B(7)
                .
                .
                .
```

In example 1, the CONTINUE statement is used as the last statement in the range of the DO in order to avoid ending the DO loop with the statement GO TO 7.

EXAMPLE 2

```
                .
                .
                .
  DO 30 I = 1, 20
  IF (A(I)-B(I))  5, 40, 40
5 A(I) = C(I)
  GO TO 30
40 A(I) = 0.0
30 CONTINUE
                .
                .
                .
```

In example 2, the CONTINUE statement provides a branch point enabling the programmer to bypass the execution of statement 40.

4.3.6 STOP Statement

GENERAL FORM

```
STOP
```

The STOP statement terminates the execution of the object program.

4.3.7 END Statement

GENERAL FORM

```
END
```

The END statement is a nonexecutable statement that defines the end of a source program or source subprogram for the compiler. Physically, it must be the last statement of each program or subprogram. The END statement also terminates program execution if it is encountered during execution of the program.

CHAPTER 5

INPUT/OUTPUT STATEMENTS

Input/output statements are used to transfer and control the flow of data between internal storage and an input/output device, such as a TTY Terminal or disk storage unit.

5.1 INPUT/OUTPUT LIST

Input/output statements in FORTRAN are primarily concerned with the transfer of data between storage locations defined in a FORTRAN program and records external to the program. On input, data is taken from a record and placed into storage locations that are not necessarily contiguous. On output, data is gathered from diverse storage locations and placed into a record. An I/O list is used to specify which storage locations are used. The I/O list can contain variable names, subscripted array names, unsubscripted array names, or array names accompanied by indexing specifications in a form called an implied DO. No function references or arithmetic expressions are permitted in an I/O list.

If an unsubscripted array name appears in the list, the entire array is transmitted in the order in which it is stored. (If the array has more than one dimension, it is stored in ascending storage locations, with the value of the first subscript quantity increasing most rapidly and the value of the last increasing least rapidly. An example is given in Paragraph 2.7.2, "Arrangement of Arrays in Storage.")

If an implied DO appears in the I/O list, the elements of the array(s) specified by the implied DO are transmitted. The implied DO specification is enclosed in parentheses. Within the parentheses there are one or more subscripted array names, separated by commas with a comma following the last name, followed by indexing parameters $i=m_1$, m_2, m_3. The indexing parameters are as defined for the DO statement. Their range is the list of the DO-implied list and, for input lists, i, m_1, m_2, and m_3 may appear within that range only in subscripts.

For example, assume that A is a variable and that B, C, and D are 1-dimensional arrays each containing 20 elements. Then the statement:

```
PRINT 998, A, B, (C(I), I=1,4), D(4)
```

writes the current value of variable A, the entire array B, the first four elements of the array C, and the fourth element of D.

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

```
READ 998, ((A(I,J), J=1,10), B(I), I=1,20)
```

The order of the names in the list specifies the order in which the data is transferred between the record and the storage locations.

Data is transmitted under control of a FORMAT statement controlling the transmission of the data in the record from a form that can be read by the programmer to a coded form that satisfies the needs of machine representation. The transformation for input takes the character codes and constructs a machine representation of an item. The output transformation takes the machine representation of an item and constructs character codes suitable for printing. Most transformations involve numeric representations that require base conversion. The programmer must include a FORMAT statement in the program and must give the statement number of the FORMAT statement in each READ or WRITE statement.

5.2 SEQUENTIAL INPUT/OUTPUT STATEMENTS

There are four sequential input/output statements : READ, WRITE, PRINT, and REWIND. The READ and WRITE statements initiate the transfer of records of sequential files or terminal data transfer. The PRINT statement is used to transfer data to the terminal. The REWIND statement controls the positioning of the file. In addition to these four statements, the FORMAT statement, although not an input/output statement, is used with the READ, WRITE, and PRINT statements.

Before data can be read from or written to a disk file the file must be opened. When file I/O is complete the file must be closed before the program is terminated. See Paragraph 5.8, OPENF/CLOSEF Statements for a discussion of these commands

5.3 READ STATEMENT

GENERAL FORM

READ a, list
READ (b,a) list

Where: a is the statement number of the FORMAT statement describing the record(s) being read.
b is an unsigned integer constant or an integer variable that is in the range $1 \leq b \leq 255$ and represents a file reference number.
list is an I/O list of the variables that will be read in accordance with FORMAT a.
The file number 100 may be used to read from terminal.

The READ statement may take many forms. The value of a must always be specified, but under appropriate conditions b can be omitted.

The basic forms of the READ statement are:

FORM	PURPOSE
READ (b,a) list	Formatted READ from input device. If b = 100, formatted READ from a TTY terminal.
READ a, list	Formatted READ from TTY Terminal.

The form READ (b,a) list is used to read data from file number b into the variables whose names are given in the list. The data is transmitted from the file to memory according to the specifications in the FORMAT statement, which is statement number a.

EXAMPLE 1

```
READ (5,98) A,B,(C(I,K),I=1,10)
```

The above statement causes input data to be read from the data file number 5 into the variables A, B, C(1,K), C(2,K),...,C(10,K) in the format specified by the FORMAT statement whose statement number is 98.

The form READ a, list is used to read data from the TTY Terminal according to the specifications of FORMAT statement a.

EXAMPLE 2

```
READ 98, A,B,(C(I,K),I=1,10)
```

The above statement causes input data to be read from the TTY Terminal keyboard into the variables A, B, C(1,K), C(2,K),...,C(10,K) in the format specified by the FORMAT statement whose statement number is 98.

EXAMPLE 3

```
READ (100,98) A, B, (C(I,K), I=1,10)
```

The above statement reads data from the terminal as in the preceding example.

The following example shows a sample program used to read a disk file.

```
      I=10
      J='XX'
      K=0
      CALL OPENF(I,J,K,L)
      READ(I,50) II,JJ,KK
      PRINT 50, II,JJ,KK
      CALL CLOSEF(I,J,K,L)
50    FORMAT (3I3)
      END
```

5.4 WRITE STATEMENT

GENERAL FORM

WRITE (b,a) list

Where: a is the statement number of the FORMAT statement describing the record(s) being written.
b is an unsigned integer constant or an integer variable that is in the range $1 \leq b \leq 255$ and represents a file reference number.
The file number 101 may be used to print data at the terminal. The file number 102 is used to print data at the line printer.
list is optional and is an I/O list of variables that will be written to disk according to the FORMAT a.

The statement WRITE (b,a) list is used to write data into the file whose reference number is b from the variables whose names are given in the list. The data is transmitted from memory to the file according to the specifications in the FORMAT statement, whose statement number is a.

EXAMPLE

```
WRITE (10,75) A,(B(I,3),I=1,10,2),C
```

The above statement causes data to be written from the variables A, B(1,3), B(3,3), B(5,3), B(7,3), B(9,3), and C to file number 10 in the format specified by the FORMAT statement whose statement number is 75. If the file number was 101 instead of 10, the data would have been printed at the terminal, or if it were 102, data would have been printed on the line printer.

5.5 PRINT STATEMENT

GENERAL FORM

PRINT a, list

Where: a is the statement number of the FORMAT statement describing the record(s) being printed.
list is optional and is an I/O list of variables that will be printed according to the FORMAT a.

The statement PRINT a, list is used to print data at the TTY Terminal from the variables whose names are given in the list. The data is transmitted from memory to the TTY Terminal according to the specifications in the FORMAT statement, whose statement number is a.

EXAMPLE

```
PRINT 75, A, (B(I,3),I=1,10,2), C
```

The above statement causes data to be written from the variables A, B(1,3), B(3,3), B(5,3), B(7,3), B(9,3), and C to the TTY Terminal in the format specified by the FORMAT statement whose statement number is 75.

5.6 REWIND STATEMENT

GENERAL FORM

REWIND b

Where: b is an unsigned integer constant or integer variable that is in the range $1 \leq b \leq 255$ and represents a file reference/number.

The REWIND statement causes a subsequent READ or WRITE statement referring to b to read data from or write data into the first record of file number b.

5.7 FORMAT STATEMENT

GENERAL FORM

xxxxx FORMAT (c_1, c_2, ... , c_n)

Where: xxxxx is a statement number (1 through 5 digits.)
c_1, c_2, ... , c_n are format codes.

The format codes are:

aIw	Describes integer data fields.
aEw.d	Describes real data fields.
aFw.d	Describes real data fields.
aAw	Describes alphameric data fields.
'Literal'	Transmits literal data.
wX	Indicates that a field is to be filled with blanks on output.
a(...)	Indicates a group format specification.

Where: a is optional and is an unsigned integer constant used to denote the number of times the format code is to be used. If a is omitted, the code is used only once.
w is an unsigned nonzero integer constant that specifies the number of characters in the field.
d is an unsigned integer constant specifying the number of decimal places to the right of the decimal point; i.e., the fractional portion.
(...) is a group format specification. Within the parentheses are format codes separated by commas or slashes. Group format specifications can be nested to a level of two. The a preceding this form is called a group repeat count. Note: Both commas and slashes can be used as separators between format codes (see Paragraph 5.7.1, "Various Forms of a FORMAT Statement."

The FORMAT statement is used in conjunction with the I/O list in the READ, PRINT, and WRITE statements to specify the structure of FORTRAN records and the form of the data fields within the records. In the FORMAT statement, the data fields are described with format codes, and the order in which these format codes are specified gives the structure of the FORTRAN records. The I/O list gives the names of the data items to make up the record. The length of the list in conjunction with the FORMAT statement specifies the length of the record (see Paragraph 5.7.1). Throughout this paragraph, the examples show TTY Terminal input and printed line output. The concepts apply to all input/output media.

The following list gives general rules for using the FORMAT statement:

1. FORMAT statements are not executed; their function is to supply information to the object program. They may be placed anywhere in the source program after specification statements.

2. When defining a FORTRAN record by a FORMAT statement, it is important to consider the maximum size record allowed on the input/output medium. For example, if a FORTRAN record is to be printed, the record should not be longer than 72 characters.

3. If the I/O list is omitted from the READ, WRITE, or PRINT statement, a record is skipped on input, or a blank record is inserted on output.

4. Types I and A are valid only with integer variables. Types E and F are valid only with real.

5.7.1 Various Forms of a FORMAT Statement

All of the format codes in a FORMAT statement are enclosed in a pair of parentheses. Within these parentheses, the format codes are delimited by the separators: comma and slash.

Execution of a READ, WRITE or PRINT statement initiates format control. Each action of format control depends on information provided jointly by the I/O list--if one exists--and the format specification. There is no I/O list item corresponding to the format descriptors X and literals enclosed in apostrophes. These output information directly to the record.

Whenever an I, E, F, or A, code is encountered, format control determines whether or not there is a corresponding element in the I/O list. If there is such an element, appropriately converted information is transmitted. If there is no corresponding element, the format control terminates.

If, however, format control reaches the last outer right parenthesis of the format specification and another element is specified in the I/O list, control is transferred to the group repeat count of the group format specification terminated by the last right parenthesis that precedes the right parenthesis ending the FORMAT statement.

The question of whether or not there are further elements in the I/O list is asked only when an I, E, F, or A, or the final right parenthesis of the format specification is encountered. Before this is done, X, literals enclosed in apostrophes, and slashes are processed. If there are fewer elements in the I/O list than there are format codes, the remaining format codes are ignored.

5.7.1.1 COMMA. The simplest form of a FORMAT statement is the one shown at the beginning of Paragraph 5.7 with the format codes, separated by commas, enclosed in a pair of parentheses. One FORTRAN record is defined by the beginning of the FORMAT statement (left parenthesis) to the end of the FORMAT statement (right parenthesis). For an example, see Paragraph 5.7.4.

5.7.1.2 SLASH. A slash is used to indicate the end of a FORTRAN record format. For example, the statement:

```
25    FORMAT          (I3,F6.2/E10.3,F6.2)
```

describes two FORTRAN record formats. The first, third, etc., records are transmitted according to the format I3, F6.2, and the second, fourth, etc., records are transmitted according to the format E10.3, F6.2.

Consecutive slashes can be used to introduce blank output lines. If there are n consecutive slashes at the beginning or end of a FORMAT statement, n blank lines are inserted between output records. If n consecutive slashes appear anywhere else in a FORMAT statement, the number of blank lines inserted is n-1. For example, the statement:

```
25    FORMAT          (1X,10I5//1X,8E14.5)
```

describes three FORTRAN record formats. On output, it causes double spacing between the line written with format 1X,10I5 and the line written with the format 1X,8E14.5.

5.7.2 I Format Code

The I format code is used in transmitting integer data. For example, if a PRINT statement refers to a FORMAT statement containing I format codes, the input data is assumed to be stored in internal storage in integer format.

INPUT	Leading, embedded, and trailing blanks in a field of the input card are ignored.
OUTPUT	If the number of significant digits and sign required to represent the quantity in the storage location is less than w, the leftmost print positions are filled with blanks. If it is greater than w, the number is truncated.

5.7.3 E and F Format Codes

The E and F format codes are used in transmitting real data. The data must not exceed the maximum magnitude for a real constant.

INPUT — Input must be a real number which, optionally, may have an exponent. The decimal point may be omitted. If it is present, its position overrides the position indicated by the d portion of the format field descriptor, and the number of positions specified by w must include a place for it. Each data item must be right justified in its field. Leading, trailing, and embedded blanks are ignored. These two format codes are interchangeable for input. It makes no difference, for example, whether E, or F is used to describe a field containing 12.42E08.

OUTPUT — For data written under a E format code, output consists of an optional sign (required for negative values), a decimal point, the number of significant digits specified by d, and an E exponent requiring four positions. The w specification must provide for all these positions, including the one for a sign when the output value is negative. If additional space is available, a leading zero may be written before the decimal point.

For data written under an F format code, w must provide sufficient spaces for an integer segment, if it is other than zero, a fractional segment containing d digits, a decimal point, and, if the output value is negative, a sign. If insufficient positions are provided, the number is shifted and truncated. If excess positions are provided, the number is preceded by blanks.

For E and F format codes, fractional digits in excess of the number specified by d are dropped.

5.7.4 Examples of Numeric Format Codes

The following examples illustrate the use of the format codes I, F, and E.

EXAMPLE 1

```
75 FORMAT (I3,F5.2,E10.3,E10.3)

PRINT 75, N,A,B,C
```

1. Four fields are described in the FORMAT statement and four variables are in the I/O list. Therefore, each time the PRINT statement is executed, one line is printed on the TTY Terminal.

2. When a line is printed, the number in integer format in location N is printed in the first field of the line (three columns). The number in the second field of the line (five columns) is printed in real format, with no decimal exponent, and comes from location A, etc.

3. If there were one more variable in the I/O list, say M, another line would be printed and the information in the first three columns of that line would be printed in integer format and obtained from location M. The rest of the data on the line would be blank.

4. If there were one fewer variable in the list (say C is omitted), no number would be printed according to the format E10.3.

5. This format statement defines only one record format. Paragraph 5.7.1 "Various Forms of a FORMAT Statement" explains how to define more than one record format in a FORMAT statement.

EXAMPLE 2

Assume that the following statements are given:

```
76 FORMAT (F6.2,E12.3,I5)

   PRINT 76, A,B,N
```

and that the variables A, B, and N have the following values:

A	B	N
034.40	123.380E+02	031
031.10	1156.10E+02	130
0	834.621E-03	428
01.132	83.121E+06	000

Then, the following lines are printed:

34.40	0.123E+05	31
31.10	0.116E+06	130
0.00	0.835E+00	428
1.13	0.831E+08	0

5.7.5 A Format Code

The A format code is used in transmitting data that is stored internally in character format. The number of characters transmitted under A format code is limited to two characters per integer variable. The A format code may not be used with real variables. Each alphabetic or

special character is given a unique internal code. Numeric data is converted digit by digit into internal format, rather than the entire numeric field being converted into a single binary number. Thus, the A format code can be used for numeric fields, but not for numeric fields requiring arithmetic.

EXAMPLE 1

```
9900 FORMAT (A2,A1)

     READ 9900,I,K
```

Assume the following is entered at the ? when the program is executed:

```
?ABC
```

The AB will be stored in I and B will be left justified and stored in K.

Assume the above is now printed with a different FORMAT:

```
9910 FORMAT (2A1)

     PRINT 9910, I,K
```

The following will be printed at the console:

```
AC
```

EXAMPLE 2

```
     DIMENSION I(4)
     I(1) = 'MO'
     I(2) = 'TO'
     I(3) = 'RO'
     I(4) = 'LA'
     PRINT 9900, I
9900 FORMAT (4A2)
```

MOTOROLA will be printed at the console.

LITERAL DATA

Literal data can appear in a FORMAT statement as a string enclosed in apostrophes.

```
25  FORMAT  (' 1975 INVENTORY REPORT')
```

No item in the I/O list corresponds to the literal data. The data is written directly from the FORMAT statement. (The FORMAT statement can contain other types of format codes with corresponding variables in the I/O list.) For example, the following statements:

```
8 FORMAT ('MEAN AVERAGE:', F9.4)

  PRINT 8, AVRGE
```

would cause the following record to be written if the value of AVRGE were 12.3456:

```
MEAN AVERAGE:  12.3456
```

5.7.6 X Format Code

The X format code specifies a field of w characters to be skipped on input or filled with blanks on output. For example, the following statements:

```
5 FORMAT (I10,10X,4I10)

  READ   (5,5) I,J,K,L,M
```

cause the first ten characters of the input record to be read into variable I, the next ten characters to be skipped over without transmission, and the next four fields of ten characters each to be read into the variables J, K, L, and M.

5.7.7 Group Format Specification

The group format specification is used to repeat a set of format codes and to control the order in which the format codes are used.

The group repeat count a is the same as the repeat indicator a, which can be placed in front of other format codes. For example, the following statements are equivalent:

```
10 FORMAT    (I3,2(I4,I5),I6)

10 FORMAT    (I3,I4,I5,I4,I5,I6)
```

Group repeat specifications control the order in which format codes are used since control returns to the last group repeat specification when there are more items in the I/O list than there are format codes in the FORMAT statement (see Paragraph 5.7.1, "Various Forms of a FORMAT Statement"). Thus, in the previous example, if there were more than six items in the I/O list, control would return to the group repeat count 2 which precedes the specification (I4,I5).

The format codes within the group repeat specification can be separated by commas and slashes. For example, the following statement is valid:

```
40 FORMAT    (2I3/(3F6.2,F6.3/E10.3,3E10.2))
```

The first record is transmitted according to the specification 2I3, the second, fourth, etc., records are transmitted according to the specification 3F6.2,F6.3, and the third, fifth, etc., records are transmitted according to the specification E10.3,3E10.2, until the I/O list is exhausted.

5.7.8 Free Format Input

Data may be read from the TTY Terminal in free field, comma-separated input by specifying an empty format. For example,

```
998 FORMAT ( )
```

Data read in this manner will be converted to integer or real, depending upon the mode of the receiving variable. The values to be typed for the variables must be in the proper format for a real or integer constant, and are separated by commas.

EXAMPLE

```
      READ 998, I,J,X

  998 FORMAT ( )
```

The values may be input in the following form:

```
  3,5,8.3

 -3,6,5.8

  etc.
```

Alphanumeric data may be input into integer variables according to the A1 or A2 format codes. These are the only format codes that may be used for literal inputs.

5.8 OPENF/CLOSEF STATEMENTS

The OPENF and CLOSEF statements give the FORTRAN programmer control of disk file handling. Under the EDOS operating system, only one input file and one output file may be opened at a given time. With the MDOS operating system, a total of four (4) disk files can be open at a given time.

5.8.1 OPENF/CLOSEF Statement Arguments for EDOS

GENERAL FORM

```
CALL OPENF(IUNIT,IFILE,IMODE,IDRV)
CALL CLOSEF(IUNIT,IFILE,IMODE,IDRV)
```

Where: IUNIT is an unsigned integer constant or an integer variable in the range $1 \leq IUNIT \leq 255$ and represent a file reference number (FORTRAN unit number).
IFILE is a one to three element integer array containing the file name as a one to six literal ASCII character string or a single integer variable containing the file name as a one to two literal ASCII character string.
If less than six characters are specified, the file name is padded with spaces. Although six characters may be specified, only the first five are considered significant by the present operating system; therefore, the first five must be unique.
If less than five characters are specified, the last characters must be followed by blanks if IFILE is an array name.
IMODE is an unsigned integer constant or an integer variable specifying the mode with which the file is to be opened or closed.

0 = input (read)
1 = output (write)

IDRV is an unsigned integer constant or an unsigned variable specifying the disk drive unit number.

No defaults are assumed for any of the arguments, therefore, all arguments must be specified. Additional information about the arguments is in Paragraph 8.7, "Arguments in a Function or Subroutine Subprogram".

5.8.2 OPENF/CLOSEF Programming Considerations

The statement CALL OPENF(IUNIT,IFILE,IMODE,IDRV) is used to open a disk file for input(read) or output (write). To open a file for input, the file name must already exist in the directory. The directory will be searched for the file name and a fatal error condition will occur if the file is not found. To open a file for output the file name must not be in the directory. If the directory search finds the named file, a fatal error condition occurs. See Appendix C, Execution Time Error Messages.

The statement CALL CLOSEF (IUNIT,IFILE,IMODE,IDRV) is used to close a disk file after input from or output to a file is complete. Since only one input file and one output file may be opened at a given time it is necessary to close an old file before another can be opened. Failure to do so will result in a fatal error condition. In any case, all files should be closed before exiting from the FORTRAN program.

5.8.3 OPENF/CLOSEF Examples (EDOS only)

The following examples illustrate several OPENF/CLOSEF calls. All examples assume that I,K, and L have been assigned valid values in previous assignments or data statements.

EXAMPLE 1

```
CALL OPENF(I,'FN',K,L)
CALL CLOSEF(I,'FN',K,L)
```

EXAMPLE 2

```
J = 'FN'
CALL OPENF(I,J,K,L)
CALL CLOSEF(I,J,K,L)
```

EXAMPLE 3

```
CALL OPENF(I,'FLNAME',K,L)
CALL CLOSEF(I,'FLNAME',K,L)
```

This form is valid and will not cause an error; however, only the first two characters, i.e., "FL" are passed and used as the file name.

EXAMPLE 4

```
DIMENSION J(3)
DATA J/'FL','NA','ME'/
  .
  .
  .
CALL OPENF(I,J,K,L)
CALL CLOSEF(I,J,K,L)
```

EXAMPLE 5

```
DIMENSION J(3)
J(1) = 'FL'
J(2) = 'NA'
J(3) = 'ME'
  .
  .
  .
CALL OPENF(I,J,K,L)
CALL CLOSEF(I,J,K,L)
```

5.8.4 OPENF/CLOSEF Statement Arguments for MDOS

GENERAL FORM

```
CALL OPENF(IUNIT, IFILE, IMODE)
CALL CLOSEF(IUNIT)
```

Where: IUNIT is an unsigned integer constant or an integer variable in the range $1 \leq IUNIT \leq 255$, and represents a file reference number (FORTRAN unit number).

IFILE is a 1-7 element integer array containing the file name (in standard MDOS format) as a 1-13 literal ASCII character string or a single integer variable containing the file name as a 1-2 literal ASCII character string. The file name in standard MDOS format is as follows:

FILENAME.SX:N

Where "FILENAME" is a 1-8 character name, the period (".") is the suffix delimiter "SX" is a 1-2 character suffix, the colon (":") is the logical drive delimeter, and "N" is the logical drive number.

IMODE is an unsigned integer constant or an integer variable specifying the mode with which the file is to be opened.

0 = illegal mode
1 = input mode
2 = output mode
3 = update mode

No defaults are assumed for any of the arguments; therefore, all arguments must be specified. Note that three (3) arguments are required for OPENF, while only one (1) is required for CLOSEF. While no defaults are assumed for any arguments, the suffix and/or logical drive number portion(s) of IFILE will default to "SA" and "0", respectively, if omitted.

Additional information about the arguments is in Paragraph 8.7, "Arguments in a Function or Subroutine Subprogram".

5.8.5 OPENF/CLOSEF Programming Considerations for MDOS

The statement CALL OPENF (IUNIT, IFILE, IMODE) is used to open a file for input (read), output (write), or update. To open a file for input, the file name must already exist in the directory. If the file is not found in the directory, an appropriate MDOS error is returned. To open a file for output, the file name must not be in the directory. If the file name already exists, or if there is no more room in the disk directory or the disk file area, an appropriate MDOS error is returned. When opening a file for update, an open input is performed if the file is found in the directory; otherwise, an open output is performed.

The statement CALL CLOSEF (IUNIT) is used to close a disk file after input from or output to a file is complete. If the file was opened for input, a flag will be set to indicate the file is no longer open. If opened for output, an end-of-file record is written, the directory is updated, and a flag is set to indicate the file is no longer open. All files should be closed before exiting from the FORTRAN program.

5.8.6 OPENF/CLOSEF Examples for MDOS

The following examples illustrate several OPENF/CLOSEF CALLS. The examples assume that I, K, and L have been assigned valid values in previous assignment or data statements.

EXAMPLE 1:

```
CALL OPENF (I, 'FN', K)
CALL CLOSEF (I)
```

EXAMPLE 2:

```
J='FN'
CALL OPENF (I,J,K)
CALL CLOSEF (I)
```

EXAMPLE 3:

```
DIMENSION J(3)
DATA J/'FL', 'NA', 'ME'/
   .
   .
   .
CALL OPENF (I,J,K)
CALL CLOSEF (I)
```

EXAMPLE 4:

```
DIMENSION J(4)
J(1)='FI'
J(2)='LE'
J(3)='NA'
J(4)='ME'
   .
   .
   .
CALL OPENF (I,J,K)
CALL CLOSEF(I)
```

In the four examples above, the OPENF call will result in the default suffix (SA) and the default logical .drive number (0) being used.

EXAMPLE 5:

```
DIMENSION J(4)
DATA J/'F1', 'LE', '.F', 'T'/
CALL OPENF (I,J,K)
CALL CLOSEF (I)
```

EXAMPLE 6:

```
DIMENSION J(S)
J(1)='F1'
J(2)='LN'
J(3)='AM'
J(4)='E.'
J(5)='FT'
   .
   .
   .
CALL OPENF (I,J,K)
CALL OPENF (I)
```

In the two examples above, the OPENF call will result in the default logical drive number being used.

EXAMPLE 7:

```
DIMENSION J(4)
J(1)='F1'
J(2)='LE'
J(3)='NA'
J(4)=':1'
   .
   .
   .
CALL OPENF (I,J,K)
CALL CLOSEF (I)
```

EXAMPLE 8:

```
DIMENSION J(3)
DATA J/'F1', 'L:', '1'/

CALL OPENF (I,J,K)
CALL CLOSEF (I)
```

In the two examples above, the OPENF call will result in the default suffix (SA) being used.

EXAMPLE 9:

```
DIMENSION J(5)
J(1)='FL'
J(2)='NA'
J(3)='ME'
J(4)='.F'
J(5)=':1'
   .
   .
   .
CALL OPENF (I,J,K)
CALL CLOSEF (I)
```

EXAMPLE 10:

```
      DIMENSION J(7)
      DATA J/'F1', 'LE', 'NA', 'ME', '.S', 'A:', '0'/
           .
           .
           .
      CALL OPENF (I,J,K)
      CALL CLOSEF (I)
```

CHAPTER 6

DATA INITIALIZATION STATEMENT

GENERAL FORM

DATA $\underline{k}_1/\underline{d}_1,\underline{k}_2/\underline{d}_2,\ldots\ldots,\underline{k}_n/\underline{d}_n$

Where: Each $\underline{k}$ is a variable or array name. Dummy arguments may not appear in the list.
Each $\underline{d}$ is a list of constants (integer, real, or literal).
Literal data must be enclosed in apostrophes.

A DATA initialization statement is used to define initial values of variables, and arrays. There must be a one-to-one correspondence between the total number of elements specified or implied by the list $\underline{k}$ and the total number of constants specified by the corresponding list $\underline{d}$.

This statement cannot precede any other specification statement that refers to the same variables or arrays. Otherwise, a DATA statement can appear anywhere in the program but must not include variables declared to be in COMMON.

EXAMPLE

```
DIMENSION X(3)

DATA I/5/,J/-3/,X/8.0,-3.6,12.3/
```

The DATA statement indicates that the variables I, and J are to be initialized to the values 5, and -3 respectively. In addition, the statement specifies that the three variables in the array X are to be initialized to the values 8.0, -3.6, and 12.3.

CHAPTER 7

SPECIFICATION STATEMENTS

7.1 DEFINITION

The specification statements provide the compiler with information about the nature of the data used in the source program. In addition, they supply the information required to allocate locations in memory for this data.

Specification statements must precede statement function definitions, which must precede the program part containing at least one executable statement. Within the specification statements, any statement describing data must precede references to that data.

7.2 DIMENSION STATEMENT

GENERAL FORM

DIMENSION $a_1(k_1),a_2(k_2),a_3(k_3),.....,a_n(k_n)$

Where: a_1, a_2, a_3, ... a_n are array names.
k_1, k_2, k_3, ... k_n are each composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

The information necessary to allocate storage for arrays used in the source program may be provided by the DIMENSION statement. The following example illustrates how this information may be declared.

EXAMPLE

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

DIMENSION B (25,50), TABLE (5,8,4)
```

7.3 COMMON STATEMENT

GENERAL FORM

COMMON $a(k_1),b(k_2),.....c(k_3),d(k\ ),.....$

Where: $a,b,.....,c,d,.....$ are variable names or array names that cannot be dummy arguments.
$k_1,k_2,.....k_3,k$ ..... are required only if the variable represents an array name and are each composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

The COMMON statement is used to define a storage area that can be referred to by a calling program and one or more subprograms, and to specify the names of variables and arrays to be placed in this area. Therefore, variables or arrays that appear in a calling program or subprogram can be made to share the same storage locations with variables or arrays in other subprograms. Also, a COMMON area can be used to implicitly transfer arguments between a calling program and a subprogram. Arguments passed in COMMON are subject to the same rules with regard to type, length, etc., as arguments passed in an argument list (see Chapter 8, "SUBPROGRAMS").

If more than one COMMON statement appears in a calling program or subprogram, the entires in the statements are cumulative. Redundant entries are not permitted.

Since the entries in a COMMON area share storage locations, the order in which they are entered is significant. Consider the following example:

EXAMPLE

Calling Program	Subprogram
.	SUBROUTINE MAPMY (...)
.	.
.	.
COMMON A, B, C, K(100)	.
.	COMMON X, Y, Z, J(100)
.	.
.	.
CALL MAPMY (...)	.

In the calling program, the statement COMMON A, B, C, K(100) would cause 212 storage locations to be reserved in the following order:

Beginning of common area	A	B	C	Layout of storage
	4 locations	4 locations	4 locations	
	K(1)	...	K(100)	
	2 locations	...	2 locations	

The statement COMMON X, Y, Z, J(100) would then cause the variables X, Y, Z, and J(1)...J(100) to share the same storage space as A, B, C, and K(1)...K(100), respectively. Note that values for X, Y, Z, and J(1)...J(100), because they occupy the same storage locations as A, B, C, and K(1)...K(100), do not have to be transmitted in the argument list of a CALL statement.

CHAPTER 8

SUBPROGRAMS

8.1 GENERAL

It is sometimes desirable to write a program which, at various points, requires the same computation to be performed with different data for each calculation. It would simplify the writing of that program if the statements required to perform the desired computation could be written only once and then could be referred to freely, with each subsequent reference having the same effect as though these instructions were written at the point in the program where the reference was made.

For example, to take the cube root of a number, a program must be written with this object in mind. If a general program were written to take the cube root of any number, it would be desirable to be able to combine that program (or subprogram) with other programs where cube root calculations are required.

The FORTRAN language provides for the above situation through the use of subprograms. There are two classes of subprograms: FUNCTION subprograms and SUBROUTINE subprograms. In addition, there is a group of FORTRAN-supplied functions. Functions differ from SUBROUTINE subprograms in that they return at least one value to the calling program, whereas SUBROUTINE subprograms need not return any.

8.2 NAMING SUBPROGRAMS

A subprogram name consists of from one through six alphameric characters, the first of which must be alphabetic. A current EDOS limitation is five characters and must be considered if the source file is to have the same name as the subprogram contained therein. A subprogram name may not contain special characters (see Appendix A). The type of a function name determines the type of the result that can be returned from it.

The type of a FUNCTION is determined by the predefined convention for variable names.

The type of a SUBROUTINE subprogram cannot be defined because the results that are returned to the calling program are dependent only on the type of the variable names appearing in the argument list of the calling program and/or the implicit arguments in COMMON.

8.3 FUNCTIONS

A function is a statement of the relationship between a number of variables. To use a function in FORTRAN, it is necessary to:

1. Define the function (i.e., specify which calculations are to be performed).

2. Refer to the function by name where required in the program.

8.3.1 Function Definition

There are three steps in the definition of a function in FORTRAN:

1. The function must be assigned a unique name by which it may be called (see Paragraph 8.2).

2. The dummy arguments of the function must be stated.

3. The procedure for evaluating the function must be stated.

8.3.2 Function Reference

When the name of a function, followed by a list of its arguments, appears in any FORTRAN expression, it references the function and causes the computations to be performed as indicated by the function definition. The resulting quantity replaces the function reference in the expression, and assumes the type of the function. The type of the name used for the reference must agree with the type of the name used in the definitiion.

8.4 FUNCTION SUBPROGRAMS

The FUNCTION subprogram is a FORTRAN subprogram consisting of a FUNCTION statement followed by other statements, including at least one RETURN statement. It is an independently written program that is executed wherever its name is referenced in another program.

GENERAL FORM

FUNCTION name (a_1,a_2,a_3,...a_n)

Where: name is the name of the FUNCTION.
a_1,a_2,a_3, ... a_n are dummy arguments. They must be nonsubscripted variable, array, or dummy names of SUBROUTINE or other FUNCTION subprograms. (There must be at least one argument in the argument list.)

Since the FUNCTION is a separate subprogram, the variables and statement numbers within it do not relate to any other program.

The FUNCTION statement must be the first statement in the subprogram. The FUNCTION subprogram may contain any FORTRAN statement except a SUBROUTINE statement or another FUNCTION statement.

The name of the function must be assigned a value at least once in the subprogram—either as the variable name to the left of the equal sign in an assignment statement, as an argument of a CALL statement, or in an input list (READ statement) within the subprogram.

The dummy arguments of the FUNCTION subprogram (i.e., $\underline{a}_1$, $\underline{a}_2$, $\underline{a}_3$, ... $\underline{a}_n$) may be considered to be dummy variable names. These are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. Additional information about arguments is in Paragraph 8.7, "Arguments in a FUNCTION or SUBROUTINE Subprogram."

The relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the FUNCTION subprogram is illustrated in the following example:

EXAMPLES

Calling Program	FUNCTION Subprogram
	FUNCTION CALC (A,B,J)
.	.
.	.
	I = J*2
ANS = ROOT1*CALC(X,Y,I)	.
.	.
.	.
.	CALC = A**I/B
	.
	.
	.
	RETURN
	END

In this example, the values of X, Y, and I are used in the FUNCTION subprogram as the values of A, B, and J, respectively. The value of CALC is computed, and this value is returned to the calling program, where the value of ANS is computed. The variable I in the argument list of CALC in the calling program is not the same as the variable I appearing in the subprogram.

Calling Program	FUNCTION Subprogram
	FUNCTION KALC(I,J,K)
.	.
.	.
	I = J*2
ANS = ROOT1*KALC(L,M,I)	.
.	.
.	.
.	KALC = I+J+K**2
	.
	.
	.
	RETURN
	END

The FUNCTION subprogram KALC is considered an INTEGER of length 2.

8.5 RETURN STATEMENT IN A FUNCTION SUBPROGRAM

All FUNCTION subprograms must contain at least one RETURN statement.

The RETURN statement signifies a logical conclusion of the computation and returns the computed value and control to the calling program. There may be more than one RETURN statement in a FORTRAN subprogram.

EXAMPLE

```
      FUNCTION DAV (D,E,F)
      IF (D-E) 10, 20, 30
   10 A = D+2.0*E
                .
                .
                .
    5 A = F+2.0*E
                .
                .
                .
   20 DAV = A+B**2
                .
                .
      RETURN
   30 DAV = B**2
                .
                .
                .
      RETURN
```

8.6 SUBROUTINE SUBPROGRAMS

INTRODUCTION

The SUBROUTINE subprogram is similar to the FUNCTION subprogram in many respects. They both require an END and RETURN statement, and they both contain the same sort of dummy arguments. Like the FUNCTION subprogram, the SUBROUTINE program is a set of commonly used computations, but it need not return any results to the calling program, as does the FUNCTION subprogram.

The SUBROUTINE subprogram is referenced by the CALL statement, which consists of the word CALL followed by the name of the subprogram and its parenthesized arguments.

GENERAL FORM

SUBROUTINE *name* (a_1,a_2,a_3, ...,a_n)

Where: *name* is the SUBROUTINE name (see Paragraph 8.2, "Naming Subprograms").
a_1, a_2, a_3, ... a_n are dummy input and/or output arguments. (There need not be any.) Each argument used must be a nonsubscripted variable or array name.

Since the SUBROUTINE is a separate program, the variables and statement numbers within it do not relate to any other program.

The SUBROUTINE statement must be the first statement in the subprogram. The SUBROUTINE subprogram may contain any FORTRAN statement except a FUNCTION statement or another SUBROUTINE statement.

The SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. Any arguments so used must appear to the left of an arithmetic statement, in an input list within the subprogram, as arguments of a CALL statement, or as arguments in a function reference. The SUBROUTINE name must not appear in any other statement in the SUBROUTINE subprogram.

The dummy arguments (a_1, a_2, a_3, ..., a_n) may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement. Additional information about dummy arguments is in the Paragraph 8.7, "Arguments in a FUNCTION or SUBROUTINE Subprogram."

EXAMPLE

The relationship between variable names used as arguments in the calling program and the dummy variable used as arguments in the SUBROUTINE subprogram is illustrated in the following example. The object of the subprogram is to "copy" one array directly into another.

```
CALLING PROGRAM                 SUBROUTINE SUBPROGRAM
DIMENSION X(100),Y(100)
          .                     SUBROUTINE COPY (A,B,N)
          .                     DIMENSION A (100),B(100)
K = 100                         DO 10 I = 1, N
CALL COPY (X,Y,K)           10  B(I) = A(I)
          .                     RETURN
          .                     END
```

8.6.1 CALL Statement

The CALL statement is used to call a SUBROUTINE subprogram.

GENERAL FORM

CALL name (a_1,a_2,a_3,...,a_n)

Where: name is the name of a SUBROUTINE subprogram.
a_1, a_2, a_3, ... a_n are the actual arguments that are being supplied to the SUBROUTINE subprogram.

EXAMPLE

```
CALL OUT
CALL MATMPY (X,5,40,Y,7,2)
CALL QDRTIC (X,Y,Z,ROOT1,ROOT2)
```

The CALL statement transfers control to the SUBROUTINE program, and replaces the dummy variables with the value of the actual arguments that appear in the CALL statement.

8.6.2 RETURN Statement in a SUBROUTINE Subprogram

GENERAL FORM

RETURN

The normal sequence of execution following the RETURN statement of a SUBROUTINE subprogram is to the next statement following the CALL in the calling program.

8.7 ARGUMENTS IN A FUNCTION OR SUBROUTINE SUBPROGRAM

The dummy arguments of a subprogram appear after the FUNCTION or SUBROUTINE name and are enclosed in parentheses. They are replaced at the time of execution by the actual arguments supplied in the CALL statement or function reference in the calling program. The dummy arguments must correspond in number, order, type, and length to the actual arguments. For example, if an actual argument is an integer constant, then the corresponding dummy argument must be an integer. If a dummy argument is an array, the corresponding actual argument must be (1) an array, or (2) an array element. In the first instance, the size of the dummy array must not exceed the size of the actual array. In the second, the size of the dummy array must not exceed the size of that portion of the actual array which follows and includes the designated element.

*The actual arguments can be:

- Any type of constant
- Any type of subscripted or nonsubscripted variable
- An array name

If a literal constant is passed as an argument, the actual argument passed is the literal as defined, without delimiting apostrophes. A maximum of two characters can be passed as a literal.

When the dummy argument is an array name, an appropriate DIMENSION statement must appear in the subprogram. None of the dummy arguments may appear in a COMMON statement.

If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a subscripted or unsubscripted variable name, or an array name. A constant should not be specified as an actual argument unless the programmer is certain that the corresponding dummy argument is not assigned a value in the subprogram.

A referenced subprogram cannot define dummy arguments such that the subprogram reference causes those arguments to be associated with other dummy arguments within the subprogram or with variables in COMMON. For example, if the function DERIV is defined as

```
FUNCTION DERIV (X,Y,Z)
COMMON W
```

* See note Appendix D, page D-1

and if the following statements are included in the calling program

```
COMMON B
   .
   .
   .
C = DERIV (A,B,A)
```

then X, Y, Z, and W cannot be defined (e.g., cannot appear to the left of an equal sign in an arithmetic statement) in the function DERIV.

APPENDIX A

SOURCE PROGRAM CHARACTERS

Alphabetic Characters

A	N
B	O
C	P
D	Q
E	R
F	S
G	T
H	U
I	V
J	W
K	X
L	Y
M	Z

Numeric Characters

0	5
1	6
2	7
3	8
4	9

Special Characters

(blank)	*
+	,
-	(
/	' (apostrophe)
=	&
.	$
)	

The forty-nine characters listed above constitute the set of characters acceptable by FORTRAN, except in literal data, where any valid card code is acceptable.

APPENDIX B

COMPILER ERROR MESSAGES

When errors are detected by the compiler, the message:

```
***ERROR code column
```

is printed at the TTY terminal where "code" represents one of the following coded errors. "Column" represents the contents of the remainder of the line at the point that the error was detected. For example:

```
0250     IF(J-3 10,20,30

****ERROR 05 10,20,30
```

00	illegal character
01	syntax error
02	program contains too many variable names, symbol table overflow
03	statement is too complex for compiler
04	string is too long
05	syntax error
10	duplicate statement label
11	name already defined
12	array dimension too large
13	COMMON variables cannot be initiated in DATA statements
20	too many statement labels
21	function/subroutine name already used
22	dummy argument name already used
30	too many operands in this statement
31	number of subscripts does not agree with number of dimensions
50	too many nested DO's
51	one of the DO arguments is not an integer
52	index cannot be a dummy name
F0	illegal action

APPENDIX C

EXECUTION TIME ERROR MESSAGES

If a fatal error is detected during execution, the message

```
BKPT ERROR
P-xxxx   X-xxxx   A-xx   B-xx   S-xxxx
```

will be printed, where xxxx refers to the hexadecimal register contents.

The contents of the A register will contain one of the following error codes:

33	attempt to open a file for output (write) that is already open
34	disk not ready
36	duplicate file name
37	attempt to open a file for input (read) that does not exist
38	output file not open
3C	no more disk space or directory space
3D	OPENF/CLOSEF arguments must be integer (cannot be real)
40	too many repeats in the format statement
42	one input file already open The EXORciser floppy disk allows only one input file open at a time.
43	one output file already open The EXORciser floppy disk allows only one output file open at a time.
44	attempt to rewind file that is not open
B0	subscript exceeds allowed range

APPENDIX D

MOTOROLA SUPPLIED FUNCTIONS

NAME	TYPE OF ARGUMENT	TYPE OF FUNCTION	NUMBER OF ARGUMENTS	DESCRIPTION
SQRT(X)	Real	Real	1	square root of X
EXP(X)	Real	Real	1	e**X
SIN(X)	Real	Real	1	sin of X
COS(X)	Real	Real	1	cosine of X
ATAN(X)	Real	Real	1	arctangent of X
ALOG(X)	Real	Real	1	natural logarithm of X
ABS(X)	Real	Real	1	absolute value of X
POWER(X,Y)	Real	Real	2	computes X**Y
MOD(I,J)	Integer	Integer	2	computes I modulus J

NOTE: Arguments must be a single variable or constant. Expressions are not allowed.
Due to the method of using temporary storage when calculating values of functions, operations on two functions will not return the correct value.

For example:

```
A=SIN(X) + COS(Y)
```

will not return correct value to A. But:

```
A=SIN(X) + 0.0 + COS(Y)
```

does return correct value.

In addition there are several functions intended for real time use. These are:

IAND(I,J)	Integer	Integer	2	logical AND of I and J
IOR(I,J)	Integer	Integer	2	logical OR of I and J
IEOR(I,J)	Integer	Integer	2	logical EOR of I and J
ISHFT(I,J)	Integer	Integer	2	shift I left or right J position; negative J-shift right; positive J-shift left

Library error messages from the above functions are defined as follows:

LIB ERR #1

Power function cannot be called with a -X.

LIB ERR #2

Cannot take log of a negative number

LIB ERR #3

Cannot take SIN or COS of a negative number

LIB ERR #4

Cannot take square root of a negative number

The type of arguments are important because improper types may return incorrect values from the function.

APPENDIX E

EXAMPLES OF FORTRAN

This appendix illustrates features of the M6800 Resident Fortran Compiler. It will show how to build and correct a source file using the Resident Editor and then how to compile, link, and execute the program using the EDOS or MDOS system.

Steps for the complete procedure are:

a. Build source with editor or copy from tape to disk.

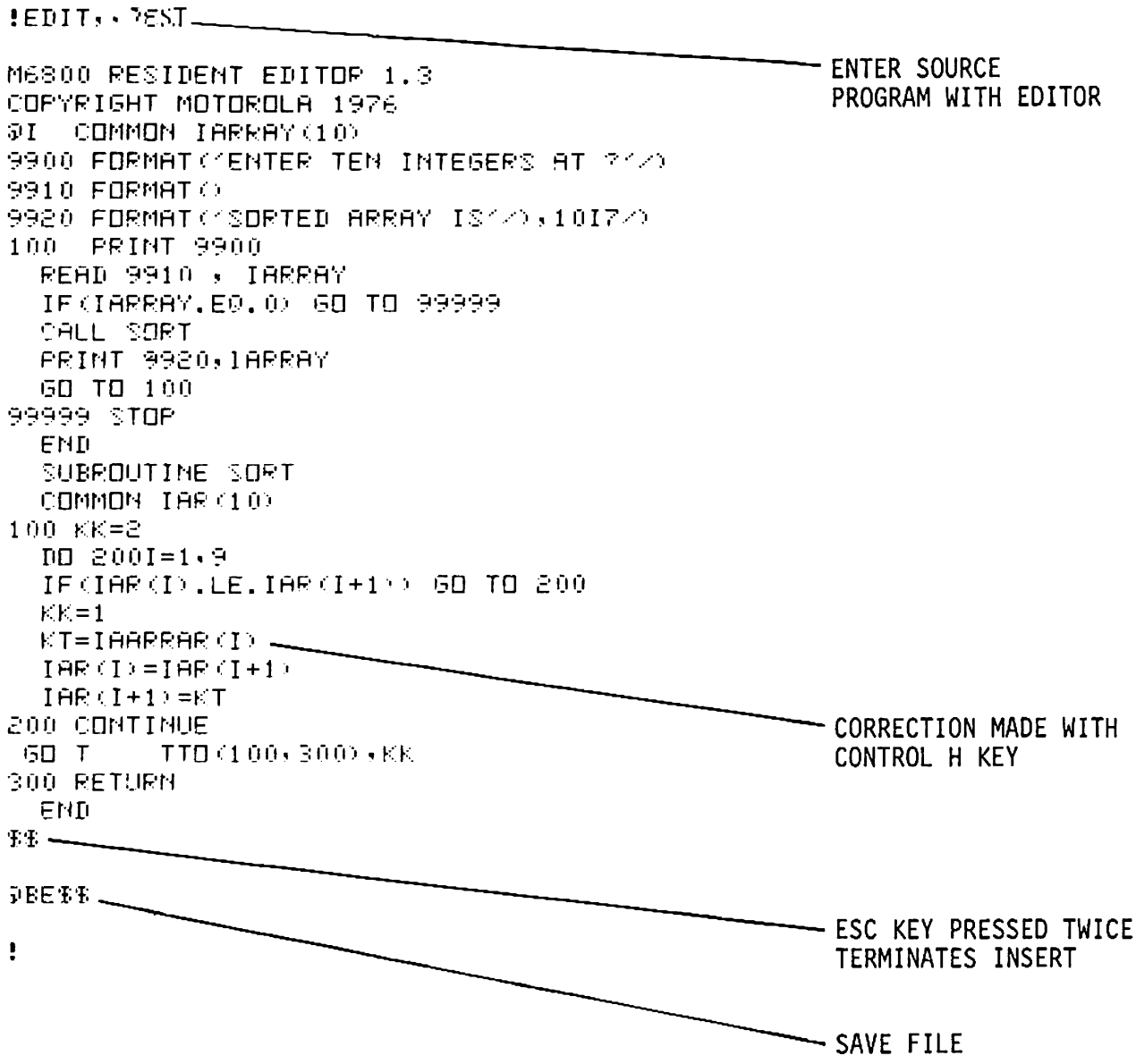
b. Compile with FORT command.

c. Link edit with RLOAD command.

d. Load module created in step c with LOAD command and execute in MAID by starting at 20;G.

The following list of EDOS commands are used in these examples. If you have an MDOS system, simply substitute the equivalent MDOS command as shown in this list.

EDOS COMMAND	EQUIVALENT MDOS COMMAND
EDIT,,SORT	EDIT SORT
FORT,#CN3,SORTO,SORT	FORT SORT;-PL=#CN,O=SORTO
PURGE,SORTO	DEL SORTO.RO
EDIT,SORT,SORTY	EDIT SORT
RLOAD	RLOAD
LOAD,SORTM 20;G	LOAD SORTM;VG
RASM,#LP,ASM1O,ASM1	RASM ASM1;LO=ASM1
MERGE,MYLIB,FORLB,PORT	MERGE FORLB,PORT,MYLIB



FOR MORE INFORMATION ON THE EDITOR, SEE THE CORESIDENT EDITOR REFERENCE MANUAL, M68CRE(D).

FIGURE E-1.

CREATING SOURCE WITH RESIDENT EDITOR

```
!FORT,#CN3,SORTO,SORT
```

COMPILE PROGRAM WITH PRINTOUT TO CONSOLE, NO PAGING

```
M6800 RESIDENT FORTRAN 1.2
COPYRIGHT BY MOTOROLA 1976


♦0000 00001          COMMON IARRAY(10)
♦0000 00002 9900   FORMAT('ENTER TEN INTEGERS AT ?'/)
♦0020 00003 9910   FORMAT()
♦0025 00004 9920   FORMAT('SORTED ARRAY IS'/),10I7/)
 ♦♦ERROR 00005
♦003D 00005 100    PRINT 9900
♦0046 00006          READ 9910,IARRAY
♦0055 00007          IF(IARRAY.EQ.0) GO TO 99999
 ♦♦ERROR 00031
♦0055 00008          CALL SORT
♦0058 00009          PRINT 9920,IARRAY
♦0067 00010          GO TO 100
♦006C 00011 99999  STOP
♦006F 00012          END

DEFINED SYMBOLS
IARRAY 0000    SORT    07FF


00002 ERRORS

♦0000 00013          SUBROUTINE SORT
♦0003 00014          COMMON IAR(10)
♦0003 00015 100    KK=2
♦000E 00016          DO 200I=1,9
♦0014 00017          IF(IAR(I).LE.IAR(I+1)) GO TO 200
♦0040 00018          KK=1
♦004B 00019          KT=IAR(I)
♦0060 00020          IAR(I)=IAR(I+1)
♦007F 00021          IAR(I+1)=KT
♦0094 00022 200    CONTINUE
♦009D 00023          GO TO(100,300),KK
♦00BA 00024 300    RETURN
♦00BB 00025          END

DEFINED SYMBOLS
KK      000E    SORT    0000    I       0012    IAR     0002    KT      001E


00000 ERRORS
!PURGE,SORTO
SORTO PURGED
PACKING DISK
```

DELETE OBJECT FILE

```
!EDIT,SORT,SORTY
```

GO TO EDITOR TO CORRECT ERRORS

```
M6800 RESIDENT EDITOR 1.3
COPYRIGHT MOTOROLA 1976
$R$$
```

FIGURE E-2.
COMPILING SOURCE

```
@S9920$0LT$$
9920 FORMAT('SORTED ARRAY IS'/),10I7/)

@C),$,$0LT$$
9920 FORMAT('SORTED ARRAY IS'/,10I7/)

@3LT$$
  IF(IARRAY.EQ.0) GO TO 99999

@CY$Y(1)$0LT$$
  IF(IARRAY(1).EQ.0) GO TO 99999
@BE$$ ————————————————————————————————— SAVE CORRECTED FILE

!FORT,#CN3,SORTO,SORTY————————————————— RECOMPILE


M6800 RESIDENT FORTRAN 1.2
COPYRIGHT BY MOTOROLA 1976


♦0000 00001          COMMON IARRAY(10)
♦0000 00002 9900   FORMAT('ENTER TEN INTEGERS AT ?'/)
♦0020 00003 9910   FORMAT()
♦0025 00004 9920   FORMAT('SORTED ARRAY IS'/,10I7/)
♦0045 00005 100    PRINT 9900
♦004E 00006          READ 9910,IARRAY
♦005D 00007          IF(IARRAY(1).EQ.0) GO TO 99999
♦0075 00008          CALL SORT
♦0078 00009          PRINT 9920,IARRAY
♦0087 00010          GO TO 100
♦008C 00011 99999  STOP
♦008F 00012          END

DEFINED SYMBOLS
IARRAY 0000     SORT    076E


00000 ERRORS

♦0000 00013          SUBROUTINE SORT
♦0003 00014          COMMON IAR(10)
♦0003 00015 100    KK=2
♦000E 00016          DO 200I=1,9
♦0014 00017          IF(IAR(I).LE.IAR(I+1)) GO TO 200
♦0040 00018          KK=1
♦004B 00019          KT=IAR(I)
♦0060 00020          IAR(I)=IAR(I+1)
♦007F 00021          IAR(I+1)=KT
♦0094 00022 200    CONTINUE
♦009D 00023          GO TO(100,300),KK
♦00BA 00024 300    RETURN
♦00BB 00025          END

DEFINED SYMBOLS
KK      000E     SORT    0000     I       0012     IAR     0002     KT      001E


00000 ERRORS
```

FIGURE E-3.

CORRECTING AND RECOMPILING SOURCE

```
!RLOAD                                    USE LINKING LOADER TO CREATE
                                          LOAD MODULE.

M6800 LINKING LOADER REV 1.1A
COPYRIGHT BY MOTOROLA 1976                REFER TO M6800 LINKING LOADER
?LOAD=SORTO                               MANUAL FOR MORE INFORMATION
?LIB=FORLB                                ON THE LINKING LOADER.
?BO=SORTM;ABSP
?LOAD=SORTO;LIB=FORLB                     LIB FORLB CONTAINS RUNTIME I/O
?MAPF                                     ROUTINES, MATH PACKAGE, AND OPENF/
  NO UNDEFINED SYMBOLS                    CLOSEF RELOCATABLE PROGRAMS AND
MAP                                       MUST BE USED TO COMPLETE THE
 S SIZE  STR  END COMN                    LINKING.
 A 0014 0026 0039
 B 0000 0000 FFFF 0000
 C 0014 003A 004D 0014
 D 0168 004E 01B5 0000
 P 143B 01B6 15F0 0000
MODULE NAME BSCT DSCT PSCT
  MAIN       0000 004E 01B6               MAIN PROGRAM
  SORT       0000 008C 0248               SUBROUTINE SORT
  RUN        0000 00AE 0303               RUNTIME I/O ROUTINES LOADED
  RXIO       0000 00D1 0BC8               FROM FORLB LIBRARY.
DEFINED SYMBOLS
 NAME  S  STR  NAME  S  STR  NAME  S  STR  NAME  S  STR  NAME  S  STR
RUN    P 0327 MAIN  P 01B6 SORT  P 0248 ENDFRW P 1116 IO1   P 0BDD
IO2    P 0CC2 IO3   P 10B4 LPUSED D 01A1 XFDOS  P 0308 XHSPCL P 0BCE
XPRT   P 0305 XSTP  P 0303 BUF   D 0104 XPDATA P 0BCB XRBYT  P 0BC8
XREAD  P 0BDA XWDISK P 0BD1 XWRT1 P 0BD4 XWRT2 P 0BD7
?EXIT                                     EXIT LOADER

!LOAD,SORTM                               LOAD THE LOAD MODULE

EXBUG 1.1 MAID                            ENTER MAID
♦20;G                                     ALWAYS START EXECUTION AT 20;G
ENTER TEN INTEGERS AT ?

?10,9,8,7,6,5,4,3,2,1
SORTED ARRAY IS
      1      2      3      4      5      6      7      8      9      10

ENTER TEN INTEGERS AT ?

?0
STOP
M6800 EDOS VER 2.6                        RETURN TO EDOS WHEN EXECUTION
                                          IS COMPLETED
!
```

FIGURE E-4.

LOADING AND EXECUTING THE PROGRAM

The following example defines and explains relative addressing.

```
!FORT,#CN3,TEST0,TEST

M6800 RESIDENT FORTRAN 1.2
COPYRIGHT BY MOTOROLA 1976


*0000 00001          COMMON A(5),I,B
*0000 00002          DIMENSION K(5),X(3)
*0000 00003 9900   FORMAT(5F10.2/5I5)
*0013 00004          DO 100I=1,5
*0019 00005 100    A(I)=I
*0037 00006          CALL ABC(K)
*003D 00007          PRINT 9900,A,K
*0052 00008          END

DEFINED SYMBOLS
A       0000     B       0016     ABC     07CF     X       001E     I       0014
K       000A


00000 ERRORS

*0000 00009          SUBROUTINE ABC(KK)
*000A 00010          DIMENSION KK(5)
*000A 00011          DO 200I=1,5
*0010 00012 200    KK(I)=I
*002E 00013          RETURN
*002F 00014          END

DEFINED SYMBOLS
ABC     0000     KK      0002     I       000C


00000 ERRORS
```

FIGURE E-5.

The first four digits following the * sign in column 1 indicate the relative address at which the statement on that line is compiled. Three possible relative addresses exist: Common SeCTion (CSCT), Data SeCTion (DSCT), and the Program SeCTion (PSCT). All variables in COMMON are in CSCT. An array in COMMON has ten information bytes in DSCT of which the first two bytes point to the address of the array in CSCT. All other variables and all constants are in DSCT. The rest of the program is in PSCT. The following table shows the variables and their relative offsets to some setion. (Refer to the symbols defined in Figure E-5.)

VARIABLE	OFFSET	SECTION
A	0000	CSCT
B	0016	CSCT
X	001E	DSCT
I	0014	CSCT
K	000A	DSCT

NOTE: K starts at 000A in DSCT. Bytes 0 through 9 are information bytes for A. Variable K also has ten bytes of information in 000A through 0013. Data for array K will be stored, starting at 0014.

Refer to the following load map to calculate the absolute location of these variables using a load map.

```
!RLOAD

M6800 LINKING LOADER REV 1.1A
COPYRIGHT BY MOTOROLA 1976
?CURP=\$100;LOAD=TEST0;LIB=FORLB
?BO=TESTM;ABSP
?CURP=\$100;LOAD=TEST0;LIB=FORLB
?MAPF
  NO UNDEFINED SYMBOLS
MAP
 S SIZE  STR  END COMN
 A 0014 0026 0039
 B 0000 0000 FFFF 0000
 C 001A 003A 0053 001A
 D 0184 0054 01D7 0000
 P 1600 0200 17FF 0000
MODULE NAME BSCT DSCT PSCT
  MAIN      0000 0054 0200
  ABC       0000 00BA 0300
  RUN       0000 00D0 0400
  RXIO      0000 00F3 0D00
DEFINED SYMBOLS
 NAME  S  STR  NAME  S  STR  NAME  S  STR  NAME  S  STR  NAME  S  STR
RUN    P 0424 MAIN  P 0200 ABC   P 0300 ENDFRW P 124E IO1   P 0D15
IO2    P 0DFA IO3   P 11EC LPUSED D 01C3 XFDOS  P 0405 XHSPCL P 0D06
XPRT   P 0402 XSTP  P 0400 BUF   D 0126 XPDATA P 0D03 XRBYT  P 0D00
XREAD  P 0D12 XWDISK P 0D09 XWRT1 P 0D0C XWRT2  P 0D0F
?EXIT

!
```

FIGURE E-6.

NOTE: CSCT is $1A_{16}$ bytes long and goes from $3A_{16}$ to 53_{16}. Therefore, A starts at $3A_{16}$, while B is at $3A_{16}+16=50_{16}$. DSCT for MAIN (which is really TEST) starts at 54_{16}. The first ten bytes are information for A and K starts at $54_{16}+A_{16}=5E_{16}$. The main program (TEST) starts execution at 0200_{16} because that is where PSCT begins. The statement CALL ABC(K) can be found at $200_{16}+37_{16}=237_{16}$, while the DO statement starts at $200_{16}+13_{16}=213_{16}$. The return statement in ABC can be found at $300_{16}+2E_{16}=32E_{16}$.

APPENDIX F

LINKING FORTRAN AND ASSEMBLY LANGUAGE PROGRAMS

This appendix illustrates the method of linking an assembly language program to a FORTRAN program. Note that after assembly, the assembly language programs are object files by themselves, but would work as well if they were in a library.

Example 1 links a main FORTRAN program and assembly language subprogram together using the data section.

Example 2 links a main FORTRAN program and assembly language program together using COMMON.

NOTE: To the FORTRAN runtime routines (RXIO and RUN) odd addresses in DSCT (data section) and CSCT (common section) have a unique meaning. Therefore, you must always have an even number of bytes reserved for both DSCT and CSCT, or you must load assembly language routines after all FORTRAN programs are loaded.

EXAMPLE 1

```
!FORT,#LP3,EX10,EX1



*0000 00001 C    THIS IS AN EXAMPLE OF LINKAGE BETNEEN AN
*0000 00002 C    M6800 ASSEMBLY LANGUAGE PROGRAM AND AN M6800 FORTRAN SOURCE
*0000 00003 C    PROGRAM  THE ASSEMBLY PROGRAM WILL TAKE THE FIRST TWO
*0000 00004 C    ARGUMENTS, THE 2 CONSTANTS, AND STORE THEM IN THE FOLLOWING
*0000 00005 C    ARGUMENTS, THE 2 VARIABLES, K AND C
*0000 00006 C
*0000 00007          CALL ASM1(300,20.0,K,C)
*000F 00008          PRINT 9900,K,C
*0024 00009 9900   FORMAT('K=',I5,'  C=',F6.1)
*0036 00010          END

DEFINED SYMBOLS
ASM1    07FF     C       0008     K       0006


00000 ERRORS
```

FIGURE F-1.

COMPILE FORTRAN PROGRAM

```
!RASM,#LP,ASM1O,ASM1


PAGE  001   ASM1

00001                                   NAM     ASM1
00002                                   OPT     REL
00003                                   XDEF    ASM1      DEFINE INTERNAL SYMBOL
00004                                   XREF    RUN  ,    EXTERNAL SYMBOL 'RUN'
00005                           *
00006                           *  THE FOLLOWING VARIABLES MUST BE PUT IN THE
00007                           *  DATA SECTION, AND MUST BE 2 BYTES LONG.  THEY
00008                           *  WILL HOLD ADDRESS OF EACH ARGUMENT IN THE
00009                           *  CALL WHICH WILL BE SET UP AT JSR RUN+15
00010                           *
00011D 0000                             DSCT
00012D 0000     0002  A I       RMB     2
00013D 0002     0002  A AA      RMB     2
00014D 0004     0002  A L       RMB     2
00015D 0006     0002  A CC      RMB     2
00016                           *
00017                           *  START THE PROGRAM SECTION HERE
00018                           *
00019P 0000                             PSCT
00020                           *  THIS IS THE ENTRY POINT FROM THE
00021                           *  FORTRAN PROGRAM
00022P 0000 BD 000F   A ASM1    JSR     RUN+15    SET UP LINKAGE AT THIS
00023P 0003     0000  D         FDB     I         ADDRESS IN RUNTIME PACKAGE
00024P 0005     0002  D         FDB     AA        TO ADDRESS IN CALLING PROG
00025P 0007     0004  D         FDB     L         THERE MUST BE AN FDB HERE
00026P 0009     0006  D         FDB     CC        FOR EVERY ARGUMENT IN CALL
00027P 000B     0000  A         FDB     0         END LIST WITH ZERO.
00028P 000D FE 0000   D         LDX     I
00029P 0010 A6 00     A         LDAA    0,X
00030P 0012 E6 01     A         LDAB    1,X
00031P 0014 FE 0004   D         LDX     L
00032P 0017 A7 00     A         STAA    0,X
00033P 0019 E7 01     A         STAB    1,X
00034P 001B FE 0002   D         LDX     AA
00035P 001E A6 00     A         LDAA    0,X
00036P 0020 E6 01     A         LDAB    1,X
00037P 0022 FE 0006   D         LDX     CC
00038P 0025 A7 00     A         STAA    0,X
00039P 0027 E7 01     A         STAB    1,X
00040P 0029 FE 0002   D         LDX     AA
00041P 002C A6 02     A         LDAA    2,X
00042P 002E E6 03     A         LDAB    3,X
00043P 0030 FE 0006   D         LDX     CC
00044P 0033 A7 02     A         STAA    2,X
00045P 0035 E7 03     A         STAB    3,X
00046P 0037 39                  RTS
00047                           END
TOTAL ERRORS 00000
```

FIGURE F-2.

```
!RLOAD

M6800 LINKING LOADER REV 1.1A
COPYRIGHT BY MOTOROLA 1976
?LOAD=EX1O,ASM1O;LIB=FORLB
?BO=EX1M;ABSP
?LOAD=EX1O,ASM1O;LIB=FORLB
?EXIT

!LOAD,EX1M

EXBUG 1.1 MAID
♦20;G
K=  300 C=  20.0
STOP
M6800 EDOS VER 2.6

!
```

FIGURE F-3.

LINK BOTH PROGRAMS AND EXECUTE
THE LOAD MODULE

EXAMPLE 2

```
!FORT,#LP3,EX2D,EX2
```

```
*0000 00001 C
*0000 00002 C  THIS IS AN EXAMPLE OF LINKING A FORTRAN AND ASSEMBLY
*0000 00003 C  LANGUAGE PROGRAM TOGETHER USING VARIABLES IN COMMON
*0000 00004 C
*0000 00005          COMMON I(10),K
*0000 00006          CALL ASM2
*0003 00007          PRINT 9900,I,K
*0018 00008 9900   FORMAT(10I5)
*0023 00009          END

DEFINED SYMBOLS
ASM2    0738     I       0000     K       0014


00000 ERRORS
```

FIGURE F-4.

COMPILE FORTRAN PROGRAM

```
!RASM,#LP,ASM20,ASM2


PAGE  001   ASM2

00001                          *
00002                          *  THIS PROGRAM USES COMMON SECTION(CSCT)
00003                          *  WHICH WILL OVERLAY THE 'COMMON' VARIABLES
00004                          *  IN A CALLING FORTRAN PROGRAM
00005                          *
00006                                    NAM     ASM2
00007                                    OPT     REL         SAVE THE RELOCATABLE CODE
00008                                    XDEF    ASM2        INTERNAL SYMBOL DEFINITION
0000 9C 0000                             CSCT                BEGIN COMMON SECTION HERE
0000 10C 0000      0014   A I            RMB     20
0000 11C 0014      0002   A K            RMB     2
0000 12P 0000                            PSCT                START PROGRAM SECTION HERE
0000 13P 0000 4F                 ASM2    CLRA
0000 14P 0001 5F                         CLRB
0000 15P 0002 CE 0000   C                LDX     #I
0000 16P 0005 5C                 LOOP    INCB
0000 17P 0006 C1 0A      A               CMPB    #10
0000 18P 0008 2E 08 0012                 BGT     DONE
0000 19P 000A A7 00      A               STAA    0,X
0000 20P 000C E7 01      A               STAB    1,X
0000 21P 000E 08                         INX
0000 22P 000F 08                         INX
0000 23P 0010 20 F3 0005                 BRA     LOOP
0000 24P 0012 B7 0014   C DONE           STAA    K
0000 25P 0015 F7 0015   C                STAB    K+1
0000 26P 0018 39                         RTS
00027                                    END
TOTAL ERRORS 00000
```

FIGURE F-5.

ASSEMBLE THE ASSEMBLY LANGUAGE PROGRAM

```
!RLOAD

M6800 LINKING LOADER REV 1.1A
COPYRIGHT BY MOTOROLA 1976
?LOAD=EX2O,ASM2O;LIB=FORLB
?BO=EX2M;ABSP
?LOAD=EX2O,ASM2O;LIB=FORLB
?EXIT

!LOAD,EX2M

EXBUG 1.1 MAID
◆2O;G
    1    2    3    4    5    6    7    8    9   10
   11
STOP
M6800 EDOS VER 2.6

!
```

FIGURE F-6.

LINK BOTH PROGRAMS AND EXECUTE

The following example shows the method used to call a FORTRAN subroutine from assembly language.

```
        XREF    SUBP
        JSR     SUBP        GO TO SUBROUTINE
        FCB     NN
        FDB     AAAA        ADDRESS OF ARGUMENT
```

NN	AAAA
00	Integer constant address
80	Floating point constant address
90	Floating point common variable address
91	Floating point common variable array
81	Dimensional floating point variable
11	Integer common variable array
01	Integer dimensional variable
00	Integer variable
80	Floating point variable
C1	Dummy dimensional floating point variable
41	Dummy dimensional integer variable
40	Dummy integer variable
C0	Dummy floating point variable

NOTE: AAAA always points to the address of the argument. NN and AAAA must be present for each argument required by the subroutine being called.

EXAMPLE:

Suppose you wanted to call a FORTRAN subroutine to print the value of XX. You must place the value in XX before the JSR.

```
        XREF    PRNT
XX      RMB     4
         .
         .
        JSR     PRNT
        FCB     $80
        FDB     XX
```

```
00001                                    NAM     CR
00002                                    OPT     REL
00003                                    XREF    SQRT,P
00004D 0000                              DSCT
00005D 0000     0140   A XX              FDB     $140      FLOATING POINT 4.0 IN XX
00006D 0002     0000   A                 FDB     0
00007P 0000                              PSCT
00008P 0000 CE 0026    A                 LDX     #$26      TEMP ADDRESS OF 4 BYTES
00009                                    *  FOR SQRT TO RETURN VALUE
00010                                    *  REQUIRED ONLY FOR FUNCTION CALL
00011                                    * TEMP CAN BE ANY 4 BYTES, DOES
00012                                    * NOT HAVE TO BE $26.
00013P 0003 BD 0000    A                 JSR     SQRT
00014P 0006     80     A                 FCB     $80
00015P 0007     0000   D                 FDB     XX
00016                                    * VALUE RETURNED IN ABS
00017                                    * LOC 26-29, PUT IT IN XX.
00018P 0009 DE 26      A                 LDX     $26
00019P 000B FF 0000    D                 STX     XX
00020P 000E DE 28      A                 LDX     $28
00021P 0010 FF 0002    D                 STX     XX+2
00022P 0013 BD 0000    A                 JSR     P
00023P 0016     80     A                 FCB     $80
00024P 0017     0000   D                 FDB     XX
00025P 0019 3E                           WAI
00026                                    END
TOTAL ERRORS 00000
```

```
      SUBROUTINE P(X)
      PRINT 9,X
    9 FORMAT(E16.8)
      RETURN
      END
```

FIGURE F-7.

CALLING FORTRAN SUBPROGRAMS

APPENDIX G

CREATING A LIBRARY ON THE EDOS SYSTEM

A library is created with the MERGE command. Consider how programs call one another before merging. For instance, if program A calls program B, program A must be merged first. Otherwise, the library must be searched twice. See the M6800 Linking Loader manual for more details.

For example, suppose you want to put a subroutine called PORT in FORLB.

```
!MERGE,MYLIB,FORLB,PORT
```

Program PORT now follows all of the FORTRAN library in a library called MYLIB.

APPENDIX H

CHANGING RUNTIME I/O ADDRESSES

It is sometimes desirable to execute FORTRAN programs on a non-resident system. In order to do this, the EXBUG I/O addresses must be changed to those of the system where the program will be executed. Following is an example of how to change these addresses.

```
00001                                   NAM     SETADD
00002                                   OPT     REL
00003                                   XREF    XRBYT,XPDATA,XWDISK
00004                                   XREF    XWRT1,XWRT2,XREAD
00005                                   XREF    XSTP,XPRT,XFDOS
00006                                   XREF    RUN,XHSPCL
00007                                   XDEF    SETADD
00008                             *
00009                             * THESE ADDRESS OFSETS ARE VALID FOR RESIDENT
00010                             * FORTRAN RELEASE 1.2.
00011                             *
00012                             * THIS ROUTINE SHOWS HOW TO OVERLAY THE EXBUG
00013                             * I/O ADDRESSES IN THE RUNTIME PACKAGE WITH
00014                             * THE USERS ADDRESSES WHEN THEY DIFFER FROM EXBUG
00015                             * CALL THIS ROUTINE FROM THE FORTRAN SOURCE
00016                             * PROGRAM BEFORE ANY I/O IS DONE IN THE FORTRAN
00017                             * PROGRAM AS FOLLOWS:
00018                             *       CALL SETADD
00019                             *
00020                             * WHEN THE RUNTIME ROUTINE ARRIVES AT ANY OF THE
00021                             * ADDRESSES THE INDEX REGISTER CONTAINS THE
00022                             * MEMORY ADDRESS OF THE DATA TO BE HANDLED.
00023                             * ON INPUT THE ADDRESS IS WHERE THE DATA WILL BE
00024                             * STORED.  ON OUTPUT WHERE THE DATA IS STORED.
00025                             * ACCUMULATOR A CONTAINS THE CHARACTER FOR
00026                             * OUTPUT TO CONSOLE AT F018.  IF IT IS A STRING
00027                             * OF DATA, AS WHEN GOING TO F024, THE STRING ENDS
00028                             * WITH 04.
00029                             *
00030                             * TO WRITE YOUR OWN I/O ROUTINES OBSERVE THE
00031                             * FOLLOWING RULES:
00032                             * ACCUMULATOR A AND B SHOULD BE SAVED.
00033                             * THE USER SHOULD PROVIDE HIS OWN STACK, SAVING
00034                             * AND RESTORING THE RUNTIME STACK.
00035                             *
00036              0000  P SETADD EQU     *       ENTRY POINT FROM FORTRAN
00037P 0000 CE 0000  A         LDX     #$0     YOUR STACK POINTER
00038P 0003 FF 0000  A         STX     XSTP    FF8A-EXBUG STACK POINTER
00039P 0006 CE 0000  A         LDX     #$0     YOUR CONSOLE CR/LF ADDRESS
00040P 0009 FF 0001  A         STX     XPRT+1  F024-CONSOLE CR/LF ADDRESS
00041P 000C FF 0001  A         STX     XPDATA+1 F024 AGAIN
00042P 000F CE 0000  A         LDX     #$0     RETURN TO YOUR DOS
00043P 0012 FF 0001  A         STX     XFDOS+1 E800-EDOS RETURN
00044P 0015 CE 0000  A         LDX     #$0     YOUR INPUT CHAR FROM DISK
00045P 0018 FF 0001  A         STX     XRBYT+1 E809-INPUT CHAR FROM DISK
00046P 001B CE 0000  A         LDX     #$0     OUTPUT LINE CR/LF TO PRINT
00047P 001E FF 0001  A         STX     XHSPCL+1 EAD5-LINE,CR/LF, PRINTER
00048P 0021 CE 0000  A         LDX     #$0     DISK OUTPUT CHAR
00049P 0024 FF 0001  A         STX     XWDISK+1 E80C-DISK OUTPUT CHAR
00050P 0027 CE 0000  A         LDX     #$0     CR/LF
00051P 002A FF 0001  A         STX     XWRT1+1 F021-CR/LF
00052P 002D CE 0000  A         LDX     #$0     PRINT CONSOLE CHAR
00053P 0030 FF 0001  A         STX     XWRT2+1 F018-PRINT CONSOLE CHAR
00054P 0033 CE 0000  A         LDX     #$0      INPUT CHAR FROM CONSOLE
00055P 0036 FF 0001  A         STX     XREAD+1 CONSOLE INPUT CHAR-F015
00056P 0039 39                 RTS
00057                          END
```

FIGURE H-1.

EXAMPLE FOR RESIDENT FORTRAN RELEASE 1.2

```
00001                                NAM     SETADD
00002                                OPT     REL
00003                           *
00004                           * THESE ADDRESS OFSETS ARE VALID FOR RESIDENT
00005                           * FORTRAN RELEASES AFTER 1.2.
00006                           *
00007                           * THIS ROUTINE SHOWS HOW TO OVERLAY THE EXBUG
00008                           * I/O ADDRESSES IN THE RUNTIME PACKAGE WITH
00009                           * THE USERS ADDRESSES WHEN THEY DIFFER FROM EXBUG
00010                           * ALL I/O ADDRESSES HAVE BEEN PUT IN A NAMED
00011                           * COMMON AREA TO PRESERVE RAM/ROM DICHOTOMY AND
00012                           * STILL ALLOW THE USER TO MODIFY THE ADDRESSES.
00013                           * PREPARE A TABLE AS THIS EXAMPLE SHOWS TO
00014                           * CHANGE THEM TO YOUR OWN.
00015                           * NOTE**** IT IS IMPORTANT THIS ASSEMBLY PROGRAM
00016                           * BE LOADED AFTER THE FORTRAN LIBRARY IS LOADED
00017                           * SO THESE ADDRESSES WILL OVERLAY THE ADDRESSES
00018                           * LOADED FROM THE LIBRARY OUT OF ROUTINE 'RUN'.
00019                           *
00020                           * WHEN THE RUNTIME ROUTINE ARRIVES AT ANY OF THE
00021                           * ADDRESSES THE INDEX REGISTER CONTAINS THE
00022                           * MEMORY ADDRESS OF THE DATA TO BE HANDLED.
00023                           * ON INPUT THE ADDRESS IS WHERE THE DATA WILL BE
00024                           * STORED.  ON OUTPUT WHERE THE DATA IS STORED.
00025                           * ACCUMULATOR A CONTAINS THE CHARACTER FOR
00026                           * OUTPUT TO CONSOLE AT F018.  IF IT IS A STRING
00027                           * OF DATA, AS WHEN GOING TO F024, THE STRING ENDS
00028                           * WITH 04.
00029                           *
00030                           * TO WRITE YOUR OWN I/O ROUTINES OBSERVE THE
00031                           * FOLLOWING RULES:
00032                           * ACCUMULATOR A AND B SHOULD BE SAVED.
00033                           * THE USER SHOULD PROVIDE HIS OWN STACK, SAVING
00034                           * AND RESTORING THE RUNTIME STACK.
00035                           * TO USE THIS TABLE YOU SIMPLY REPLACE
00036                           * EACH $0 WITH YOUR I/O ADDRESS.  FOR INSTANCE, TO
00037                           * CHANGE PRINT CONSOLE CHAR, F018, TO ADDRESS
00038                           * 3FFF CHANGE THE $0 AT XWRT2 TO $3FFF.
00039                           *
00040N 0000                    .ADDR   COMM    PSCT      .ADDR IS NAME OF COMMON
00041                          * AND MUST BE CALLED .ADDR
00042N 0000      0000  A XSTF  FDB     $0        FF8A-EXBUG STACK POINTER
00043N 0002 7E   0000  A XRBYT JMP     $0        E809-INPUT CHAR FROM DISC
00044N 0005 7E   0000  A XPDATA JMP     $0        F024-CONSOLE CR/LF ADDRESS
00045N 0008 7E   0000  A XHSPCL JMP     $0        EAD5-LINE,CR/LF TO PRINTER
00046N 000B 7E   0000  A XWDISK JMP     $0        E80C-DISC OUTPUT CHAR.
00047N 000E 7E   0000  A XWRT1 JMP     $0        F021-CR/LF
00048N 0011 7E   0000  A XWRT2 JMP     $0        F018-PRINT CONSOLE CHAR
00049N 0014 7E   0000  A XREAD JMP     $0        F015-CONSOLE INPUT CHAR
00050N 0017 7E   0000  A XFDOS JMP     $0        E800-EDOS RETURN
00051                          END
```

FIGURE H-2.

EXAMPLE FOR RESIDENT FORTRAN AFTER RELEASE 1.2

The same technique described in the two previous EDOS examples also applies to MDOS. Listed below is a jump table that must be modified when used with MDOS.

```
                XREF    BUF     BUFFER IN RXIO

.ADDR           COMM    PSCT    .ADDR MUST BE USED
                JMP     $0      CONSOLE INPUT
                JMP     $0      CONSOLE OUTPUT
                JMP     $0      LINE PRINTER
                JMP     $0      DISK INPUT
                JMP     $0      DISK OUTPUT
                JMP     $0      REWIND FILE
                JMP     $0      RETURN TO MDOS
                JMP     $0      ERROR ROUTINE
                END
```

NOTE: On input, the data must be put into the external reference BUF (a buffer in the RXIO). The data must be ASCII and must be terminated with a 4.

On output, the data will be in the external reference buffer (BUF) in ASCII, terminated with a 4.

MOTOROLA Integrated Circuit Division

microsystems • 3102 North 56th Street • Phoenix, Arizona 85018