MOTOROLA

M68FTN(D3)

M6800/M6809

MDOS FORTRAN REFERENCE MANUAL

MICROSYSTEMS

M68FTN(D3)

SEPTEMBER 1980

M6800/M6809

MDOS FORTRAN

REFERENCE MANUAL

The information in this document has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies. Furthermore, Motorola reserves the right to make changes to any products herein to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights or the rights of others.

EXORciser, EXORterm, EXORdisk, MDOS, and Micromodule are trademarks of Motorola Inc.

Third Edition

Copyright 1980 by Motorola Inc.

Second Edition March 1977

TABLE OF CONTENTS

Page

LIST OF TABLES

CHAPTER 1

MDOS FORTRAN COMPILER

1.1 INTRODUCTION

The Motorola 6800/6809 MDOS FORTRAN compiler is designed for the solution of small to medium scale scientific problems and control applications. The system consists of computer hardware and software. There are two phases to any FORTRAN program - the COMPILATION phase and the EXECUTION or RUN-TIME phase.

For the compilation phase, the minimum configuration is:

- EXORciser or EXORterm Development System
- 48K bytes of memory
- EXORdisk drive with MDOS disk operating system
- ASCII terminal (may be EXORterm above)
- MDOS Editor, FORTRAN Compiler, FORTRAN Run-time Library, and Linking Loader

The minimum compile phase configuration may be expanded to include more memory, up to four disk drives, and a variety of line printer and CRT terminal devices.

This version of FORTRAN is written to support both EXORciser/EXORterm and Micromodule configurations during the execution phase of a FORTRAN program. The object code produced by the compile/link process may be stored on diskette or may be burned into PROM or EROM. If no disk I/O is required at execution time, neither EXbug nor MDOS is required for execution of a FORTRAN program.

In addition, MDOS FORTRAN easily interfaces with assembly language programs or routines, which are assembled as relocatable modules. The object code output of the compiler is also in the form of relocatable modules.

1.2 APPLICATION

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.

With extensions incorporated into MDOS FORTRAN, many control-type applications become practical, including certain real-time applications. Three versions of MDOS FORTRAN are offered: a standard 6800 version; a standard 6809 version; and a 6800 version incorporating real-time features, including a real-time operating system. Except where indicated, this manual applies to all versions. Chapters 9 and 11 apply only to the 6800 Real-Time version.

1.3 COMPILE PHASE OPERATION

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.

The compiler analyzes the source program statements and translates them into a machine language output called object programs, which are relocatable modules. If the FORTRAN compiler detects errors in the source statements, it produces the appropriate diagnostic error message. The linking loader is utilized to create an absolute object program that can be executed by an MC6800 or MC6809 microprocessor (depending upon the version of MDOS FORTRAN being used).

1.3.1 MDOS Command Line

The MDOS FORTRAN compiler is invoked by the FORT command. This command and its parameters are defined as follows:

COMMAND NAME: FORT

PURPOSE: The FORT command processes source program statements written in MDOS FORTRAN language. These source statements are compiled into object programs by the FORTRAN compiler. Under option control, a source listing is also produced.

GENERAL FORM: FORT [<delim><sfile>{.<suffix>][:<log drv>]][;<options>]}

where: <delim> is a valid command line delimeter

<sfile> may be one or more source program files (20 max.)

<suffix> is the file name suffix (.SA if not specified)

<log drv> is the logical drive number of <sfile> (:0 if not specified)

<options> may be one or more of the compiler options shown in paragraph 1.3.2. Certain options are defaulted to being automatically specified or turned on. These options may be disabled or turned off by preceding the option letter with a minus sign (-).

1.3.2 Command Line Options

OPTION LETTER	ATTRIBUTE CONTROLLED BY OPTION	DEFAULT
A	Listing contains relative address	-A
B	Listing contains line sequence number	B
C	Source input is in card image format	-C
F	Fast subscript evaluation without error check	-F
G	Special option for Micromodule 12 (see Chapter 11)	-G

OPTION LETTER	ATTRIBUTE CONTROLLED BY OPTION	DEFAULT
H	Input initial heading from console	-H
I	All variable names are integer	-I
L	Print listing on line printer	-L
L=#LP,	Print listing on line printer (same as L)	-L
L=#CN,	Print listing on system console	-L
L=<fn>,	Print listing on disk file with name "fn" (Default suffix ".FL", default drive same as drive for first source file)	-L
M	Micromodule 14 or 14A in final system (Effective for 6800 Real-Time verson only)	-M
N=nnn,	Specifies maximum number of columns printed (50 <= nnn <= 120)	N=80
O	Produce object output in <sfilel.RO>	O
O=<fn>,	Produce object output in "fn" (Default suffix ".RO", default drive same as drive for first source file)	O
P=nn,	Specifies number of lines per page printed (10 <= nn <= 72) -P will inhibit paging	P=58
R	Compile for RAM/ROM dichotomy	-R
S	Symbol table listing	-S
X	Conditional compilation of "X" statements	-X
Y	Conditional compilation of "Y" statements	-Y

EXAMPLES:

```
=FORT CONVRT:1;LSA
```

will cause compilation of source file CONVRT.SA on drive 1, producing a compilation listing on the line printer with a symbol table, relative addresses and sequence line numbers, and an object file CONVRT.RO on drive 1.

```
=FORT PROG1.FS,PROG2.FS;L=#CN,S-O
```

will cause compilation of source files PROG1.FS and PROG2.FS on drive 0, with a compilation listing on the console with a symbol table and sequence line numbers, but no relative addresses will be displayed and no object output file will be produced.

1.3.3 Option H - Header Line

This option allows entering of a header line up to 32 characters, which will be displayed at the top of each printed page and also placed into the identification record of any relocatable object file produced.

1.3.4 Console Source Code Input

In addition to disk file source input, MDOS FORTRAN allows use of the console device for the compiler source. This may be specified by #CN in place of source disk file names. Console and disk sources may NOT be intermixed.

If an object file name is not specified, the default file name will be CN.RO on drive 0.

1.4 SOURCE LINE COMPILER DIRECTIVES

In addition to the options which may be specified on the command line when calling up the compiler, certain options and printing directives are available if embedded in the source program. All are invoked by the use of a dollar sign ($) in column 1 of the source file, and will not be printed on the compiler listing.

Directive	Meaning
$-L	Stop listing output.
$L	Start listing output again (will not override an "-L" option on the command line).
$P	Page to top of new page.
$n	Skip "n" lines on the listing, where n is 1 to 9.
$H	Change header to the 32 characters following the $H.
$G	Turn on "G" option (see Chapter 11, M68MM12).
$-G	Turn off "G" option.

1.5 OPTION STATEMENT

This statement in a FORTRAN program unit directs the compiler to change certain parameters. The options implemented at this time include processor stack size control and integer only compilation.

GENERAL FORM: OPTION a1[,a2,....,aN]

where: a1 through aN are one or more of the following:

```
STACK = value
SSTACK = value    (6809 only)
USTACK = value    (6809 only)
INTEGER
```

"value" is a decimal or hexadecimal constant whose value is the desired stack size in bytes of a main program unit.

OPTION INTEGER has the same effect as the "I" option letter on the compiler command line.

The OPTION statement(s) should be the first statement in the source file (even before a SUBROUTINE, FUNCTION, TASK, or PROGRAM statement).

The default stack sizes are 100 bytes for the SP or S stack, and 32 bytes for the U stack (6809 only).

EXAMPLES:

```
OPTION STACK=$40,INTEGER
OPTION USTACK=200,SSTACK=$80
```

1.6 COMPILER MESSAGE FILE

When invoked by the FORT command, the compiler searches for a file named FORTMSG.SA on drive 0. If it finds such a file, the contents of that file will be printed on the console output device.

This may be utilized by the user to insert any message or warning desired. To eliminate this sign on message, delete or change the name of FORTMSG.SA.

1.7 NON-LOWER CASE COMPATIBLE TERMINALS

The FORTRAN compiler normally prints some messages in upper and lower case ASCII. Since some older terminals cannot accept lower case ASCII, a special flag byte can be changed to force all messages to upper case ASCII only. This has no effect on user-entered lower case, such as might be entered in FORMAT statements or comments. The flag byte can be changed as follows:

```
=PATCH   FORT.CM
2000   20
>A/XX                where XX = 00 to enable lower case
>Q                         XX = FF to disable lower case
=
```

1.8 INCLUDE Statement

The INCLUDE statement allows calling in another source file at any point in the original source. The INCLUDE statement may be used any number of times, but may NOT be nested (one Included file calling another).

GENERAL FORM: INCLUDE 'filename'

where: filename is the MDOS source file name enclosed in apostrophes and including any needed suffix and drive number.

The default suffix is SA and the default drive is 0, and both will prevail unless explicitly stated within the apostrophes. Only one file name may appear with each INCLUDE statement.

Users will find this statement quite useful in programs consisting of many subprograms with a large COMMON. The COMMON area may be kept in a separate source file and INCLUDE'd in each subprogram as needed. This ensures that all common declarations will be the same.

EXAMPLES:

```
INCLUDE 'CBLOCK.SA:1'
INCLUDE 'ENDPROC:1'
INCLUDE 'COMMENTS'
```

While nesting is not permitted, "chaining" is. If an INCLUDE'd file contains the INCLUDE statement, it will be the last read from that file. The next file designated by the new INCLUDE will start supplying source lines to be compiled. When the end of file is reached with an INCLUDE'd file, source input reverts back to the original (command line) source file.

No special ending is used for an INCLUDE file.

CHAPTER 2

ELEMENTS OF THE FORTRAN LANGUAGE

2.1 INTRODUCTION

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. EXTERNAL declarations, if any.
3. COMMON and DIMENSION statements, if any. They may be intermixed.
4. EQUIVALENCE statements, if any.
5. DATA statements, if any.
6. Executable statements.
7. 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.

2.2 STATEMENTS

Source programs consist of a set of statements from which the compiler generates machine instructions, constants, and storage areas. A given FORTAN 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 are composed of certain key words used with 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 line printer, disk, or other device.

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 with the main program as one source file or as a separate file not existing with the main program.

No more than one statement may appear on each source line, although one statement may occupy more than one source line through a continuation, as described in paragraphs 2.3.1 and 2.3.2

2.3 CODING FORTRAN STATEMENTS

2.3.1 Free Format Input

The statements of a FORTRAN source program can be entered with an editor on a terminal. 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.

To improve readability, as many blanks as desired may be written between keywords and variable names. 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.

If the letter C or an asterisk (*) is placed in column 1, comments for documentation purposes may be written in columns 2 through 72 of a line. Comments may appear between FORTRAN statements; a comment line may not immediately precede a continuation line. Comments are ignored by the FORTRAN compiler except for listing. Comments may also be placed on a statement line if preceded by a semicolon (;).

The "C" or "*" indicating a comment record, the "&" signifying statement continuation, the "$" for compiler directives, and an "X" or "Y" for conditional compilation must start in column 1. If an "X" is in column 1, the record is treated as a comment unless an "X" appears on the MDOS command line as an option when invoking the compiler. In this case, records with an "X" in column 1 will be compiled. The same is true of a "Y" in column 1 and "Y" on the command line. Statement numbers ranging from 1 to 99999 also start in column 1 and are followed by at least one blank. All other statements may start anywhere from 2 to 72.

2.3.2 Card Image Format Input

A "C" option specified on the command line when invoking the compiler allows the use of traditional "card image" type of input from a source file. With this, column 1 is the same in respect to the "C" or "*" for comments, "$" for compiler directives, and "X" or "Y" for conditional compilation. Columns 1 to 5 are otherwise used for statement labels (numbers) and column 6 for continuation. Statements must begin in column 7 or higher. Any non-blank character in column 6 will signify a continuation to the compiler.

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; literal constants may be a string of alphanumeric and/or special characters.

2.4.1 Integer Constants

An integer constant is a whole number written without a decimal point. It occupies two bytes of memory. The allowable range is +32767 to -32768 and it is interpreted as a base 10 (decimal) number. It must not contain embedded commas.

EXAMPLES:

Valid Integer constants:

```
    0
   91
  173
-21474
```

Invalid Integer Constants:

```
   27.      (contains a decimal point)
51459       (exceeds the allowable range)
5,396       (contains an embedded comma)
```

2.4.2 Hexadecimal Integer Constants

This version of FORTRAN permits the use of hexadecimal (base 16) constants wherever constants are permitted, if the constant is prefixed with the dollar sign ($). Also refer to FORMAT edit character "Z" for hexadecimal I/O.

EXAMPLE:
```
KKA = $FCF4
CALL BO($8008,$3A)
```

2.4.3 Real Constants

A real constant has one of three forms: a basic real constant, a basic real followed by a decimal exponent, or an integer constant followed by a decimal point. A real constant occupies four bytes of memory and is an approximation of a number. The precision using four bytes is approximately 6 decimal digits.

A basic real constant is a string of up to eight decimal digits with a decimal point.

The magnitude range of a real constant is 16E-64 through 16E63 (approximately 10E75), and including zero.

A real constant may be positive, zero, or negative (if unsigned, it is assumed to be positive) and must be within 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:

```
-999.9999
    7.0E0
     7.E3
    7.0E3
    7E-03
```

Invalid Real Constants:

```
        1  (Missing decimal point or decimal exponent)
3,471.1    (Embedded comma)
    1.E    (Missing an integer constant after the E)
23.5E97    (Magnitude outside the allowable range)
```

2.4.4 Literal Constants

A literal constant is a string of alphanumeric and/or special characters enclosed in apostrophes.

The string may contain any character. Each character requires one byte of storage. The number of characters in the string, including blanks, may not be greater than 72. In order to specify an apostrophe within the string, two apostrophes in succession must be used.

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

EXAMPLES:

```
'IT''S HERE!'
'X-COORDINATE   Y-COORDINATE   Z-COORDINATE'
'3.14'
K = 'AB'
```

2.5 SYMBOLIC NAMES

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

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")
- A MAIN program unit

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

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 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 (Bytes)
Integer	2
Real	4

2.6.3 Type Declaration

Type declaration by predefined specification is a convention used to specify variables as integer or real 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.

The only execption to this convention in MDOS FORTRAN is that ALL names can be declared INTEGER by use of the "I" option at compile time or OPTION INTEGER at the beginning of the source program.

2.7 ARRAYS

A FORTRAN array is a set of variables identified by a single variable name. A particular variable in the array may be referred top 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, the 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 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 a 3 dimensional 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

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 plus or minus integer constant.

The following restrictions apply to the construction of subscript quantities:

1. Subscript quantities may not contain arithmentic 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 - e.g., 3, 21, 418

2. Integer variable

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

EXAMPLES:

Valid Subscripted Variables:

```
ARRAY(IHOLD)
NEXT(19)
MATRIX(I-5)
```

Invalid Subscripted Variables:

```
ARRAY(-5)    (Subscript may not be negative)
LOT(0)       (Subscript may never be zero)
ALL(X)       (Subscript 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*2
A	Real variable	REAL*4
3.14E3	Real constant	REAL*4
SIN(X)	Real function reference	REAL*4
(A*B+C)	Parenthesized real expression	REAL*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	Meaning
**	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction

2.9.3 Construction of Arithmetic Expressions

Following are the rules for construction of 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	3rd
Addition and subtraction	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
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

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

2.10 KEYWORDS

The following keywords are reserved by MDOS FORTRAN, and may not be used for any naming convention such as Symbolic names, Variable names, Array names, etc.

AND	END	IF	READ
CALL	ENDIF	INCLUDE	RETURN
CLOSE	ENDFILE	INTEGER	REWIND
COMMON	EQ	LE	SSTACK
CONTINUE	EQUIVALENCE	LT	STACK
DATA	EXTERNAL	NE	STOP
DECODE	FORMAT	NOT	SUBROUTINE
DIMENSION	FUNCTION	OPEN	TASK
DO	GE	OPTION	THEN
ELSE	GO	OR	TO
ELSEIF	GOTO	PRINT	USTACK
ENCODE	GT	PROGRAM	WRITE

In addition, future releases of MDOS FORTRAN may implement some or all of the following list. Therefore, these names should be avoided in user programs if they are expected to be upward compatible:

ASSIGN	CONSTANT	ON
BACKSPACE	DOUBLE	PAUSE
BIT	ERROR	PRECISION
BLANK	IMPLICIT	REAL
BLOCK	LOGICAL	SAVE
BYTE	NULL	STATUS
CHARACTER	OFF	ZERO

2.11 COMMENTS

As mentioned in paragraph 2.3, a source line may be a comment line by placing an asterisk (*) or the letter C in column 1.

A second method is available to add a comment after a statement on the same line. The semicolon (;) will cause the compiler to stop scanning the line; therefore, any material on the line after the semicolon will be treated as a comment.

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.

TABLE 3-1. Conversion Rules for 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.

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.

3.2 ASSIGNMENT STATEMENTS

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

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

Then the following examples illustrate valid arithmetic statments 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 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.
J = J+1	The value of J is replaced by the value of J+1.
A = C*D	The product of C and D replaces the value of A.

Multiple assignments are not permitted. As an example, A=B=C=0.0 is not permitted in MDOS FORTRAN.

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 chapter 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

GO TO XXXX

where: XXXX represents an executable statement number.

GO TO may be separated by a blank or written as GOTO.

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 (x1, x2, ..., xn) i

where: i is a nonsubscripted integer variable
n has a range: 1<=n<=20

GO TO may be separated by a blank or written as GOTO

This statement causes control to be transferred to the statement numbered x1, x2, x3,..., or xn, 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 that it is within the range 1 <= i <= n. If the i is outside that range, execution will continue at statement following the computed GOTO. No error message will be given.

EXAMPLE:

```
      GOTO (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) x1, x2, x3

where: a is any arithmetic expression.
x1, x2, x3 are any executable statement numbers.

The arithmetic IF statement causes control to be transferred to the statement numbered x1, x2, or x3 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 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 statement s will be executed if the named variable is positive (greater than or equal to zero). The variable will be considered false and 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. If the expression involves any amount of computation, a very small value is more likely to result than a zero. For this reason, IF statements using real numbers should not be programmed to have a zero or .EQ. value.

4.3.3 Block IF Statement

An alternate extension to the Logical IF statement is the block IF statement. The block IF statement is used with the END IF statement and, optionally, with the ELSE or ELSE IF statements to form a structured programming sequence of execution.

GENERAL FORM: IF (a) THEN

where: a is any logical expression.

The statement(s) following the THEN are executed if the expression a is true; otherwise, the statement following the optional ELSE or ELSE IF is executed. If no ELSE or ELSE IF statement is present, then the statement following the END IF statement is executed next if the expression is false. The statement or statements following the THEN are executed until the ELSE or END IF is encountered, then control passes to the statement following the END IF.

Block IF statements may be nested. It is important, however, to have an END IF statement paired with every IF - THEN combination.

The ELSE IF key word may contain the space, or may be written as ELSEIF. The remainder of the logical IF must continue on the same line as the ELSE IF (or on a following continuation line).

No other statements or key words may follow the THEN on a line.

The ELSE statement is used alone on a line, and there may not be any other key word following it (with the exception of the ELSE IF).

The END IF statement is used alone on a line and may be written ENDIF.

EXAMPLE:

```
IF (A.GT.B) THEN
    C=3.44
    D=C*A+6.21
  ELSE
    C=4.15
    D=C*B+7.07
  END IF
```

Note the use of indentation to aid in depicting the various levels of logic.

4.4 DO LOOPS

4.4.1 DO Statement

GENERAL FORM:

	End of Range	DO Variable	Initial Value	Test Value	Increment
DO	x	i =	m1,	m2[,	m3]

where: x is an executable statement number appearing after the DO statement.

i is a nonsubscripted integer value and cannot be a dummy.

m1, m2, and m3 are either unsigned integer constants greater than zero, or unsigned nonsubscripted integer variable whose value is greater than zero. m2 may not exceed 32767 in value. m3 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 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; each succeeding time, i is increased by the value m3. When, at the end of the iteration, i is equal to the highest value that does not exceed m2, 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:

$$\left[\frac{m2 - m1}{m3}\right] + 1$$

The brackets represent the largest integral value not exceeding the value of the expression within the brackets. If m2 is less than m1, the statements in the range of the DO are executed once.

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.

EXAMPLE 3

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

In Example 3, 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, 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.

4.4.2 Programming Considerations

1. The indexing parameters of a DO statement (i, m1, m2, m3) 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                        | Range of
        DO 50 J = 1, 5 ---| Range of          | outer DO
  50    C(J+1) = A(I)  ---| inner DO          |
                          -------------------------
```

EXAMPLE 2

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

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

4. Never transfer into the middle of a DO loop with a GO TO.

5. 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, m1, m2, m3) 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.

6. The indexing parameters (i, m1, m2, m3) may be changed by the 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.

7. 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.

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

4.5 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, arithmetic 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, 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)
    GOTO 30
40  A(I)=0.0
30  CONTINUE
    C=A(3)+B(7)
        .
        .
        .
```

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

4.6 STOP Statement

GENERAL FORM: STOP

The STOP statement defines the logical end of an executing program. Its execution causes the FORTRAN program to print the word "STOP" on the console terminal and return to the operating system. This statement may be used any number of times in a program or sub-program or may be omitted.

4.7 END Statement

GENERAL FORM: END

The END statement is a non-executable 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 replaces a STOP statement at the physical end in a program or replaces a RETURN statement at the physical end of a sub-program.

CHAPTER 5

INPUT/OUTPUT STATEMENTS

5.1 INTRODUCTION

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

5.2 INPUT/OUPUT LIST

Input/output statements in FORTRAN are primarily concerned with the transfer of data between storge 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 processed by ascending storage locations. 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 paremeters i=m1, m2, m3. 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, m1, m2, and m3, may appear within that range only in subscripts.

Example: A is a variable; B, C, and D are 1-dimension arrays, each containing 20 elements. 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 output. Most transformations involve numeric representations that require base conversion. For formatted I/O 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.

5.3 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 console terminal data transfer. The PRINT statement is used to transfer data to the console 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 5.11, OPEN/CLOSE Statement Arguments, for a discussion of these.

The following reference chart indicates the MDOS FORTRAN pre-assigned file reference number:

NUMBER	ASSIGNMENT or USAGE
99	Dummy device. Buffer I/O
100	Console keyboard
101	Console printer or display
102	Line printer
103	Reserved

5.4 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 to 255 and represents a file reference number.

list is an I/O list of the variables.

The READ statement may take two forms. The value of a must always be specified, but b can be omitted. The form READ a, list is used to read data from the console according to the specifications of FORMAT statement a.

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(J,K),J=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.

EXAMPLE 2

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

The above statement causes input data to be read from the console 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(J,K),J=1,10)
```

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

Refer to Paragrah 5.9.1 for further disk file information.

REREAD CAPABILITY: Sometimes it is desired to have records in a data file which are not uniform in format. This feature allows a re-read of the I/O record buffer without reading in a new record. Use file number 99 to accomplish this.

EXAMPLE:
```
READ(7,900)A,B,J
READ (99,901)C,K,L
```

Allows reading from file number 7 under format number 900 and rereading the same record under format number 901.

5.5 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 to 255 and represents a file reference number.
- 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(J,3),J=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 were 101 instead of 10, the data would have been printed at the console; or if it were 102, data would have been printed on the line printer.

5.6 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 console from the variables whose names are given in the list. The data is transmitted from memory to the console according to the specifications in the FORMAT statement, whose statement number is a.

EXAMPLE

```
PRINT 75,A,(B(J,3),J=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 console in the format specified by the FORMAT statement whose statement number is 75.

5.7 ENCODE/DECODE STATEMENTS

These statements are used to re-format data which is being stored in variables. ENCODE allows writing to a buffer under format control a list of variables, the same as a WRITE statement except that the characters remain in the buffer and not sent to an output device. DECODE then allows reading of that buffer under a different format control. It is much the same as a READ statement except that the characters are already in a buffer and therefore no access of an input device is required.

GENERAL FORM:

```
ENCODE fsn,list

DECODE fsn,list
```

where: fsn is the FORMAT Statement Number

list is the variable list

MDOS FORTRAN uses the I/O buffer which has a maximum length of 132 characters for ENCODE/DECODE operations. Therefore, the format statement must not contain any slash characters or exceed the maximum buffer length. Since this buffer is shared with other I/O, the DECODE statement should immediately follow the ENCODE statement in the program.

EXAMPLE:

```
      I='AB'
      J='CD'
      ENCODE 1,I,J
      DECODE 2,A
1     FORMAT(2A2)
2     FORMAT(A4)
```

In the above example, the variable "A" will contain the literal "ABCD" after execution of the statements. In the following example, a numeric integer is changed to a literal (that is, variable K contains the numeric 16-bit representation of the number 73, while L will contain the ASCII characters $37 and $33 after execution).

EXAMPLE:

```
      K=73
      ENCODE 3,K
      DECODE 4,L
3     FORMAT(I2)
4     FORMAT(A2)
```

5.8 EOFTST (END-OF-FILE TEST) SUBROUTINE

This subroutine is used to test for END-OF-FILE conditions on files. Normally, a read encountering an END-OF-FILE terminates the run in an error condition.

GENERAL FORM: CALL EOFTST(IUNIT,IFLAG)

where: IUNIT is an unsigned integer constant or an integer variable n the range 1<=IUNIT<=255, and represents a file reference number (FORTRAN UNIT NUMBER) to be tested for an end-of-file condition. The numbers 99 through 103 are reserved for special use.

IFLAG is an integer variable which is set to two (2) if an END-OF-FILE has been encountered; otherwise, it is set to one (1).

EXAMPLE:

```
      DIMENSION IN(7),IOUT(7)
      DATA IN/'DISKDATA.SA:1'/
      DATA IOUT/'DISK:1'/
      OPEN (10,IN,1)
      OPEN (11,IOUT,2)
      CALL EOFTST(10,MN)
40    READ (10,100) I,J,K,L
```

```
100  FORMAT(4I3)
     CALL EOFTST(10,MN)
     GO TO (50,60),MN
50   PRINT 100,I,J,K,L
     WRITE(11,100) I,J,K,L
     GO TO 40
60   CALL DELF(10)
     CLOSE (11)
     END
```

When using EOFTST, an END-OF-FILE status is maintained for each device. Thus, the test can be performed on as many different devices as desired during one program.

The first call EOFTST, which must occur after the file is opened, sets the END-OF-FILE indicator for this device to prevent the run from terminating when an END-OF-FILE condition is encountered on a READ. If this condition is encountered before the first execution of a call EOFTST, the run terminates. If the first call EOFTST in the above example is omitted and the file is empty, the run terminates.

Upon return from the second call EOFTST, MN is set to one (1) if an END-OF-FILE has not been encountered, or two (2) if an END-OF-FILE indication has been encountered.

Further attempts to read a file after the EOFTST has returned a "2" indication will result in an error condition.

5.9 REWIND STATEMENT

GENERAL FORM: REWIND b

where: b is an unsigned integer constant or integer variable that is in the range 1<b<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.

5.10 FORMAT STATEMENT

GENERAL FORM: xxxxx FORMAT (c1, c2, ... , cn)

where: xxxxx is a statement number (1 through 5 digits.)

c1, c2, ... , cn are format codes.

The format codes are:

aIw	Describes integer data fields.
aZw	Describes integer hexadecimal base data fields.
aEw.d	Describes real data fields.

aFw.d	Describes real data fields.
aAw	Describes alphanumeric data fields.
aRw	Describes alphanumeric data fields.
BN	Indicates a blank is ignored in numeric input field. (default)
BZ	Indicates a blank is a zero in numeric input field.
Bm	Describes a bit data field.
'Literal'	Transmits literal data.
wX	Indicates that a field is to be filled with blanks on output or skipped on input.
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 5.10.1, "Various Forms of a FORMAT Statement").

m is a bit mask.

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 edit codes, and the order in which these edit 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 5.8.1). Throughout this paragraph, the examples show console input and output. However, the concepts apply to all input/output media.

The following list gives general rules for using FORMAT statements:

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 80 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, Z, and B are valid only with integer variables. Types E and F are valid only with real variables.

5.10.1 Various Forms of a FORMAT Statement

All of the edit codes in a FORMAT statement are enclosed in a pair of parentheses, within which the edit 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 edit specification. There is no I/O list item corresponding to the edit descriptors X and literals enclosed in apostrophes. These output information directly to the record.

Whenever an I, E, F, Z, B, R 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. Format control terminates when these codes are encountered and there is no corresponding data item in the I/O list.

If, however, format control reaches the last outer right parenthesis of the edit specification and another element is specified in the I/O list, control is transferred to the group repeat count of the group edit 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, Z, B, R, or A, or the final right parenthesis of the edit 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 edit codes, the remaining edit codes are ignored.

5.10.1.1 COMMA

The simplest form of a FORMAT statement is the one shown at the beginning of Paragraph 5.10.5 with the edit 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).

5.10.1.2 SLASH

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

```
25   FORMAT (I2,F6.2/E10.3,F6.2)
```

describes two FORTRAN record formats. The 1st, 3rd, etc. records are transmitted according to the format I2, F6.2, and the 2nd, 4th, 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:

```
30   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.

5.10.1.3 Printing of Formatted Records

Format (carriage) control characters are special characters placed in the first output buffer character position to control the printing device. These format control characters are shown below:

```
(blank) - Normal CR, LF prior to printing the line.
   0    - CR, LF, LF (double spacing)
   1    - CR, LF, FF (Form Feed)
   +    - suppression of CR and LF (continues on same line)
(other) - normal CR, LF sequence prior to printing complete line including
          the character in control position.
```

Note: The complete line, including the first character, will be output to a disk data file. The control characters mentioned above have no effect when writing to disk.

EXAMPLE:

```
9000 FORMAT(' NORMAL CR,LF')
9010 FORMAT('0DOUBLE SPACING=CR,LF,LF')
9020 FORMAT('1FORM FEED,CR,LF')
9030 FORMAT('+SUPPRESSION OF CR,LF')
9040 FORMAT('ALSO NORMAL CR,LF')
9050 FORMAT(1X,I2,3A2,'NORMAL CR,LF')
9060 FORMAT('1',I5,Z3,'FORM FEED,CR,LF')
```

5.10.2 I Edit Code

The I edit code is used in transmitting integer data. For example, if a PRINT statement refers to a FORMAT statement containing I edit 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 record are ignored unless a BZ has been specified previously in the FORMAT statement, in which case all blanks are treated as zeros.

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 (except where BN has been specified, the positions will be zero filled). If it is greater than w, the number is printed and expanded to the right (w is overridden).

5.10.3 Z Edit Code

The Z edit code is the same as the I edit code, except that numeric data is interpreted as hexadecimal instead of decimal. On fields wider than necessary to print the number, leading zeros will be output. For example, with a Z4 specification, the hexadecimal number 3C4 will be printed as 03C4.

5.10.4 E and F Edit Codes

The E and F edit 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 an 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 should 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 should provide sufficient spaces for an integer segment, if it is other than zero, a fractional segment containing d digits, a decimal point, and a sign. If too few spaces are available, w will be overridden and the full number printed. If excess positions are provided, the number is preceded by blanks.

For E and F edit codes, fractional digits in excess of the number specified by d (see paragraph 5.10) are dropped.

Edit codes E, F, and I: If the columns required on a WRITE exceeds the specified number of columns in the format statement, FORTRAN will allow writing of the full number, altering the format to fit the number. Thus, with a format of F5.2, the value 1234.567 would be printed as 1234.56 (normally requiring a format of F7.2). Digits are not truncated. A column is required for the sign in the E, F, and I formats if it is a minus.

Left justifying numeric values on printout: By 'underformatting', it is possible to left justify numeric values due to the expanding formt width feature mentioned above. This could be quite useful in output such as:

```
There are 3 items in inventory.          (use of I1 format)
There are 9712 items in inventory.       (use of I1 format)
```

5.10.5 Examples of Numeric Format Codes

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

EXAMPLE 1

```
75 FORMAT (1X,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 console 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, 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 line would be blank.

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

EXAMPLE 2

Assume that the following statements are given:

```
76   FORMAT (1X,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
34.40	123.380E+02	31
31.10	11546.10E+02	130
0.00	834.621E-03	428
1.139	83.121E+06	0

Then the following lines are printed:

```
34.40   0.123E+05   31
31.10   0.115E+07  130
 0.00   0.834E+00  428
 1.13   0.831E+08    0
```

5.10.6 A and R Format Codes

The A and R format codes are used in transmitting data that is stored internally in character format. The number of characters transmitted under A or R format code is limited to two characters per integer variable or four characters per real variable. Each character is stored in ASCII. Numeric data is converted digit by digit into ASCII, rather than the entire numeric field being converted into a single binary number. Thus, the A and R format codes can be used for numeric fields, but not for numeric fields requiring arithmetic.

The difference between the A and R format codes is that the A code left justifies characters in storage, while the R code right justifies the characters in storage. For example, if a single ASCII character were stored in an integer variable (2 bytes), the character would actually be stored in the most significant (lower address) byte under the A format code. Unused bytes of the variable are blank filled with both A and R formats.

EXAMPLE 1

```
9900 FORMAT (A2,A1)
     READ 9900,I,K
```

The following is entered after the ? when the program is executed:

```
? ABC
```

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

If it is 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(5)
      I(1) = 'TH'
      I(2) = 'E'
      I(3) = 'QU'
      I(4) = 'IC'
      I(5) = 'K'
      PRINT 9900,I
9900  FORMAT (5A2)
```

"THE QUICK" will be printed at the console.

5.10.7 X Format Code

The X format code specifies a field of w characters to be skipped on input or filled with blanks on output.

EXAMPLE:

```
5 FORMAT (110,10X,4110)
    READ (5,5) I,J,K,L,M
```

The first ten characters of the input record are read into variable I, the next ten characters are skipped over, and the next four fields of ten characters each are read into the variables J, K, L, and M.

5.10.8 B Format Code

The B Format code allows writing of one or more bits of a byte. It is intended for use with an I/O device driver routine such as described in Chapter 10 for Micromodule 03.

On output, three bytes appear in the I/O buffer as a result of each B code encountered in the FORMAT statement. The first byte is a flag byte of $00. The second byte is the bit mask as specified by the "m" of Bm Format specification. The third byte is the least significant byte of data from the integer variable specified in the I/O list.

The intention is to have the device driver recognize the 00 flag byte as it scans the buffer, and then utilize the mask and data bytes to alter only the bits specified as "1's" in the mask byte.

As an example, let's say the variable J has the hexadecimal value of $A265. If the format specifier for this variable were B$C3, then the following would be output on a write statement to the I/O buffer:

```
00 C3 65
```

The mask of $C3 (or 11000011 in binary) specifies that only bits 0, 1, 6, and 7 be altered by the data of $65 (or 01100101 in binary). The result would be that bit 0 would be a 1, bit 1 would be a 0, bit 6 would be a 1, and bit 7 would be a 0 on the output device.

This format code will find use in interfacing with a device such as a PIA (MC6821) or Micromodule boards containing 32 I/O lines. Note that the device driver must interpret the data in the I/O buffer resulting from the use of this format code.

5.10.9 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). Example:

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

The following record is written if the value of AVRGE is 12.3456:

```
MEAN AVERAGE:  12.3456
```

The apostrophe may be included in the string by writing two successive apostrophes for each one to be included. Thus, to print "DOG'S BONE", a format string would be written: 'DOG''S BONE'

5.10.10 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 palced in front of other format codes. 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 precedies the specification (I4,I5).

The format codes within the group repeat specification can be separated by commas and slashes. 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.

5.10.11 Free Format Input

Data may be read from the console 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.
```

Free Format Input may NOT be used for alphanumeric data.

5.11 OPEN/CLOSE STATEMENTS

The OPEN and CLOSE statements give the FORTRAN programmer control of disk file handling. With the MDOS operating system, one or more disk files can be open at a given time.

5.11.1 OPEN/CLOSE Statement Arguments

GENERAL FORM:

```
OPEN (IUNIT,IFILE,IMODE)
CLOSE (IUNIT)
```

where: IUNIT is an unsigned integer constant or an integer variable in the range a IUNIT 255, and represents a file reference number (FORTRAN unit number). (99 through 103 are reserved for special use.)

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 delimiter, 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.

```
1 = input mode
2 = output mode
3 = append mode
```

No defaults are assumed for any of the arguments; therefore, all arguments must be specified. Note that three (3) arguments are required for OPEN, while only one (1) is required for CLOSE. 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 argments is in Paragraph 8.7, "Arguments in a Function or Subroutine Subprogram".

5.11.2 OPEN/CLOSE Programming Considerations

The statement OPEN (IUNIT, lFILE, IMODE) is used to open a file for input (read) or output (write). 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. To avoid fatal errors, see subroutine FILTST in Appendix E.

The statement CLOSE (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.

5.11.3 OPEN/CLOSE Examples

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

In the first four examples, the OPEN call will result in the default suffix (SA) and the default logical drive number (0) being used, since the suffix and logical drive are not explicitly provided.

EXAMPLE 1:

```
OPEN (I,'FN',K)
CLOSE (I)
```

EXAMPLE 2:

```
J='FN'
OPEN (I,J,K)
CLOSE (I)
```

EXAMPLE: 3

```
DIMENSION J(7)
DATA J/'FL','NA','ME'/
     .
     .
     .
OPEN (I,J,K)
CLOSE (I)
```

EXAMPLE 4:

```
DIMENSION J(7)
J(1)='FI'
J(2)='LE'
J(3)='NA'
J(4)='ME'
     .
     .
     .
OPEN (I,J,K)
CLOSE (I)
```

In the next two examples, the OPEN call will result in the default logical drive number being used. The suffix for Examples 5 and 6 is FT.

EXAMPLE 5:

```
DIMENSION J(7)
DATA J/'FILE.FT'/

OPEN (I,J,K)
CLOSE (I)
```

EXAMPLE 6:

```
DIMENSION J(7)
J(1)='FI'
J(2)='LN'
J(3)='AM'
J(4)='E.'
J(5)='FT'
     .
     .
     .
OPEN (I,J,K)
CLOSE (I)
```

In Examples 7 and 8, the OPEN call will result in the default suffix (SA) being used. The logical drive for these two examples is 1.

EXAMPLE 7:

```
DIMENSION K(7)
J(1)='FI'
J(2)='LE'
J(3)='NA'
J(4)=':1'
     .
     .
     .
OPEN (I,J,K)
CLOSE (I)
```

EXAMPLE 8:

```
DIMENSION J(7)
DATA J/'FI','L:','1'/

OPEN (I,F,K)
CLOSE (I)
```

The file name, file suffix, and logical drive number are provided in Examples 9 and 10.

EXAMPLE 9:

```
DIMENSION J(7)
J(1)='FL'
J(2)='NA'
J(3)='ME'
J(4)='.F'
J(5)='D:'
J(6)='1'
     .
     .
     .
OPEN (I,J,K)
CLOSE (I)
```

EXAMPLE 10:

```
DIMENSION A(4)
DATA A/'FILENAME.SA:0'/
     .
     .
     .
OPEN (I,A,K)
CLOSE (I)
```

5.12 UNFORMATTED I/O

MDOS FORTRAN allows unformatted READ and WRITE of data to and from disk files or external devices. The data will be written to the file in the same format that it is stored internally by MDOS FORTRAN.

Since the MDOS functions normally used on disk file I/O treat certain ASCII control characters as special, it is necessary to create and use a binary type file format, rather than ASCII. Therefore, to create and read binary files, these files are specified by different mode numbers (Table 5-1). Specifically, the mode number 9 is used to read binary files, mode number 10 is used to create and write to a binary file, and mode number 11 is used to append to a binary file.

Other than use of different mode numbers associated with the OPEN statement, the only other difference is the omission of the format statement number in any READ or WRITE statement.

TABLE 5-1. Disk File I/O Modes

MODE NUMBER	MODE	USE
1	Input	Space compressed ASCII file (normal)
2	Output	
3	Append	
5	Input	Non-space compressed ASCII file
6	Output	
7	Append	
9	Input	Binary file (used with unformatted I/O)
10	Output	
11	Append	

5.13 NON-SPACE COMPRESSED ASCII FILES

For certain applications, it may be desirable to create a disk file without the normal MDOS space compression. Use mode numbers 5, 6, and 7 in the OPEN statement to create and read this type of file. These numbers correspond to 1, 2, and 3, respectively, in the normal modes as described in paragraph 5.11.1.

CHAPTER 6

DATA STATEMENT

6.1 INTRODUCTION

The general form of the DATA statement is as follows:

```
DATA k1/d1/,k2/d2,...../,kn/dn/
```

where: Each k is a variable or array name. Dummy arguments may not appear in the list.

Each d is a list of constants (integer, real, or literal).

Literal data must be enclosed in apostrophes and may contain strings longer than a single storage element (2 for integer and 4 for real).

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 k and the total number of constants specified by the corresponding list 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. k cannot be a subscripted variable - i.e., ARRAY (2,5) is illegal. There is no bounds checking for the list (d) to fit inside the element k.

EXAMPLE

```
DIMENSION X(3)

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

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. The integer variable N would contain the literal SA.

EXAMPLE

```
DIMENSION B(4)

DATA B/'FILENAME.SA:2'/
```

In this example, the real array element B(1) will contain the ASCII characters "FILE", B(2) contains "NAME", B(3) contains ".SA:", and B(4) will contain the "2" and be spaced filled for the other 3 bytes.

NOTE

Use of the DATA statement with the "R" compiler option generates executable instructions which move the data from PSCT to DSCT at execution time.

CHAPTER 7

SPECIFICATION STATEMENTS

7.1 INTRODUCTION

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 the program part containing executable statements. Within the specification statements, any statement describing data must precede references to that data. The data must be defined before it is used.

7.2 DIMENSION STATEMENT

GENERAL FORM

```
DIMENSION a1(k1),a2(k2),a3(k3),....,an(kn)
```

where: a1,a2,a3,.....an — are array names.

k1,k2,k3,.....kn — are each composed of one through three unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array. The maximum size of an integer constant is 32767. However, an array of this size would exceed the available memory.

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(k1),b(k2),c(k3),.....,an(kn)
```

where: a,b,c,.....an — are variable names or array names that cannot be dummy arguments.

k1,k2,k3,...kn — 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 implictly 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, "PROGRAM UNITS").

IF more than one COMMON statement appears in a calling program or subprogram, the entries 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:

BYTES	VARIABLE
4	A
4	B
4	C
2	K(1)
196	K(2) - K(99)
2	K(100)

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(199), do not have to be transmitted in the argument list of a CALL statement.

The use of a second COMMON in the calling program, preceding the existing COMMON, would cause the above example to be incorrect.

7.4 EQUIVALENCE STATEMENT

The EQUIVALENCE statement is used to define one or more variable name(s) equivalent to another variable. The same memory storage locations will be shared by one or more variable names.

The main use of this statement would be to save on memory size needed for a particular application.

GENERAL FORM: EQUIVALENCE (a,b),(c,d)...,(x,y)

where each pair enclosed by parethesis are declared equivalent.

If either or both of the variables are dimensioned, they must have been declared prior to using in an equivalence statement.

Example: Suppose there were two arrays in a program - A and B. Let's dimension them first...

```
DIMENSION A(5),B(10)
```

Now, to make them occupy the same area in memory...

```
EQUIVALENCE (B,A)
```

or to make the 2nd element of A occupy the same memory location as the 5th element of B...

```
EQUIVALENCE (B(5),A(2))
```

Note that reversing the two elements in the above statement would be illegal since it would cause the lowest 3 elements of array B to fall lower than the start of array A.

7.5 EXTERNAL STATEMENT

This statement is used to declare a name to be an external reference rather than a variable name or subprogram name in a program unit.

GENERAL FORM: EXTERNAL n1,n2,...nN

where n1, n2, etc. are legal FORTRAN names.

After declaring external, the same name may not be used in any other way within the program unit. There are only two statements with which this name may be used - namely, OPEN and CALL - and then only as arguments.

CHAPTER 8

PROGRAM UNITS

8.1 INTRODUCTION

A program unit consists of a sequence of statements and may be either a main program or a subprogram. There may be only one main program unit, but several or no subprograms.

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 and SUBROUTINE. Functions differ from SUBROUTINE subprograms in that they return one value to the calling program, whereas SUBROUTINE subprograms need not return any.

8.2 NAMING PROGRAM UNITS

A program unit name consists of from one through six alphanumeric characters, the first of which must be alphabetic. A program unit name may not contain special characters (see Appendix A) or be a keyword (see Paragraph 2.10). The type of a function name determines the type of the result that can be returned from it 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 MAIN PROGRAM UNIT

A main program unit written in FORTRAN may call other subprogram units, but not vice-versa. MDOS FORTRAN produces a main program unit when the first non-comment statement is not a SUBROUTINE, FUNCTION, or TASK statement. The main program unit will initialize the microprocessor's stack pointer (SP in the 6800, and both S and U in the 6809) before executing any other code.

8.3.1 PROGRAM Statement

The PROGRAM statement is optional in a main program. If used, it will become the module externally defined name in place of a default "MAIN$", and may be observed on the memory map produced by the linking loader (RLOAD).

The general form is: PROGRAM name

where: name is an acceptable name as defined in Paragraph 8.2.

8.3.2 RETURN Statement

The RETURN statement is NOT permitted in a main program unit except in the case of a Real-Time FORTRAN main program unit which has called the subroutine SETRT.

8.4 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.

3. A maximum of 13 arguments is permitted. Expressions are not permitted.

8.4.1 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.

8.4.2 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 definition.

8.5 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.

8.5.1 FUNCTION Statement

GENERAL FORM: FUNCTION name(a1,a2,a3,...an)

where: name is the name of the FUNCTION.

a1,a2,a3,...an 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, and not more than 13).

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., a1, a2, a3,....an) 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 examples:

EXAMPLES

Calling Program

```
      .
      .
ANS = ROOT1*CALC(X,Y,I)
      .
      .
```

FUNCTION Subprogram

```
FUNCTION CALC(A,B,J)
      .
      .
I = J*2
      .
      .
      .
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 in the above example. The statement "RETURN" is not necessary in either of the examples.

8.5.2 RETURN Statement

FUNCTION subprograms may contain a RETURN statement, which 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, or the RETURN statement may be omitted (the END statement in this case generates the return).

EXAMPLE

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

8.6 SUBROUTINE SUBPROGRAMS

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. A maximum of 13 arguments is permitted.

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.

8.6.1 SUBROUTINE Statement

GENERAL FORM: SUBROUTINE name(a1,a2,a3,...,an)

where: name is the SUBROUTINE name (see Paragraph 8.2, "Naming Subprograms").

a1,a2,a3,....,an are dummy input and/or output arguments. (There need not be any, and maximum is 13.) 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, another SUBROUTINE statement, or a PROGRAM 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 keyword SUBROUTINE and the subroutine name must not appear in any other statement in the SUBROUTINE subprogram.

The dummy arguments (a1,a2,a3,...,an) 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 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 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.

EXAMPLE

Calling Program

```
DIMENSION X(100),Y(100)
     .
     .
K = 100
CALL COPY (X,Y,K)    10
     .
     .
```

SUBROUTINE Subprogram

```
SUBROUTINE COPY(A,B,N)
DIMENSION A (100),B(100)
DO 10 I = 1, N
B(I) = A(I)
RETURN
END
```

8.6.2 CALL Statement

The CALL statement is used to call a SUBROUTINE program.

GENERAL FORM: CALL name(a1,a2,a3,...,an)

where: name is the name of a SUBROUTINE subprogram.

a1, a2, a3,...,an are the actual arguments that are being supplied to the SUBROUTINE subprogram. (Maximum of 13)

EXAMPLE

```
CALL OUT
CALL MATMPY(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. A space may appear between the end of "name" and the parentheses starting the argument list.

8.6.3 RETURN Statement

SUBROUTINE subprograms may contain a RETURN statement, which signifies a logical conclusion of the computation and returns control to the calling program. There may be more than one RETURN statement in a SUBROUTINE subprogram or the RETURN statement may be omitted (the END statement in this case generates the return).

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 SUBPROGRAMS

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.

Following is an example of the actual argument being an array element:

```
Calling Program          SUBROUTINE Subprogram

DIMENSION A(6)           SUBROUTINE SAM(B)
     .                   DIMENSION B(2)
     .                        .
CALL SAM(A(5))                .
     .                        .
     .                   RETURN
                         END
```

In the foregoing example, the portion of the actual array that follows and includes the designated element is element 5 and element 6. Therefore, dummy array B must not be larger than two.

The actual arguments can be:

- Any type of constant
- Any type of subscripted or nonsubscripted variable
- An array name
- An externally declared 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 programmmer is certain that the corresponding dummy argument is not assigned a value in the subprogram.

CHAPTER 9

6800 REAL-TIME FORTRAN

9.1 INTRODUCTION

The Real-Time features available in the MDOS REAL TIME FORTRAN version give the user the capability of writing real-time software in a high-level language for ultimate use in an EXORciser, EXORterm, Micromodule or equivalent 6800 based system.

The Real-Time version not only is a language compiler, but also contains an execution-time operating system with several queues of tasks to be performed, along with an ability to respond to real-time interrupts and generation of delays.

9.2 REAL-TIME OPERATING SYSTEM

The operating system may be looked upon as having several features, namely:

Task queues
Priorities
Interrupt handling
Delay queuing

9.2.1 Task Queues

There are five queues of tasks to be performed:

1. An active queue
2. A 10-millisecond timer queue
3. A one-second timer queue
4. A one-minute timer queue
5. An interrupt association queue

Tasks or segments of tasks which are to be executed after specified time delays are placed in the 10 millisecond, 1 second, and 1 minute queues with associated counts of time delay units. The programmer can do this with calls to START and WAIT subroutines as described later.

The operating system determines when tasks are to be transferred to the active queue based upon the specified time delays. Tasks with no time delays are entered in the active queue directly.

When a READ, PRINT, or WRITE statement is encountered, the operating system does not permit the system to be locked in the I/O operation as is the case in standard FORTRAN. The operating system will start the next ready task in the active queue if a delay is encountered in waiting for an I/O device to become ready. After the device has become ready, control will return to the I/O task.

9.2.2 Priorities

Associated with each task in the active queue is a priority level. There are two classes of priorities: Immediate and Normal. Priority levels are numbered from 1 to 255. Immediate class priority levels are 1-127, while Normal class priority levels are 128-255. The lower the number, the greater the priority.

In either class, when a task is placed in the active queue with the same priority class as the currently executing task, the current task will not immediately be suspended, regardless of its priority level. Instead, the newly invoked task must wait until the current task terminates or is delayed or performs standard FORTRAN I/O. However, a task invoked with a priority in the immediate class will cause a task with priority in the normal class to be temporarily suspended until the task in the immediate class has completed execution. A task with a normal priority cannot cause the suspension of a task with immediate priority.

It is suggested that immediate class priorities only be used for short tasks requiring very high priority, since they actually interrupt the execution of a normal priority task and data integrity may be lost if data is common to both tasks.

9.2.3 Interrupt Handling

A special form of a subroutine subprogram, called a TASK, is written to perform the desired operations upon receiving an interrupt from some external device in the system.

The association between a particular interrupt and a named TASK is made with a subroutine CALL to ATTACH. Arguments passed with the call include the TASK name, the memory address of the interrupting peripheral device, a mask to determine source of the interrupt and type of device, and the priority level number of the TASK.

A given TASK can be attached to handle more than one peripheral device. The real-time operating system will prevent the same task from executing simultaneously for two or more interrupts.

9.2.4 Delay Queuing

Tasks can be invoked in either of two manners. One by external interrupt as depicted above, the other by placing it into the queue by a subroutine call to START or STARTV.

The call to START (and STARTV) requires specification of the TASK name and an associated delay. The task is placed in the appropriate timing queue (or into the active queue in case of zero delay). STARTV allows passing of two additional arguments, one of them being a priority. START uses the current executing task priority level for the priority of the newly invoked task.

Delay control routines enable the currently executing task to be suspended from execution for a period of time or until some event occurs. This suspension allows other tasks to be executed. The subroutine names for this feature are WAIT and WAITE, whereby the first specifies a time value, and the second specifies an argument which must reach a value of zero before control is returned.

CAUTION

TIME DELAY MAY NOT BE ABSOLUTELY ACCURATE DUE TO (1) ONE TASK MUST WAIT UNTIL ANOTHER IS SUSPENDED FROM OPERATION, AND (2) POSSIBLE 0 TO -1 TIME UNIT VARIATION. (EXAMPLE: ONE-MINUTE DELAY COULD BE NEAR 0 TO 60 SECONDS.) FOR THIS REASON, IT WOULD BE WISE TO USE THE SMALLEST TIME UNIT POSSIBLE.

9.3 INVOKING REAL-TIME FEATURES

A call to a subroutine in the REAL-TIME FORTRAN library is necessary to set up certain parameters, storage areas, and links to the Real-Time operating system.

9.3.1 SUBROUTINE SETRT

This subroutine is used to initialize the Real-Time system. It must be the first executable statement in a real-time program.

GENERAL FORM: CALL SETRT(a,c,m,p)

Where:

- a is the memory address of the console or control ACIA
- c is the base memory address of the PTM (programmable timer module, MC6840) or PIA (peripheral interface adapter, MC6820/MC6821) used for the real-time clock.
- m is the mask for PTM ($01) or PIA ($80 or $40) clock interrupt bit.
- p is the name of an array used for queue data pool.

An example of this would be a system containing an ACIA at $FCF4, a Micromodule MM01D board with PTM at address $EC18, and the array named KPOOL is used for a data pool. The following would be the proper call to SETRT:

```
CALL SETRT($FCF4,$EC18,1,KPOOL)
```

The execution of SETRT causes the designated PIA or PTM to be initialized to generate the proper clock interrupts to the microprocessor. During program operation, the real-time executive checks for the source of an interrupt. If it was the clock, the executive will take care of resetting the interrupt flag in the PIA or PTM.

9.3.2 QUEUE ARRAY

The queue data pool array is used to provide space for dynamic storage requirements of the real-time operating system. Storage entries are 10 bytes long (5 integer variables elements of an array). Queue entries are used for:

1. ATTACH calls.
2. START calls.
3. WAIT calls.
4. Outstanding TASKs awaiting completion.

The statement to allocate the necessary storage could be:

```
DIMENSION KPOOL(5,20)
```

if it were determined that 20 entries would be the maximum ever used at one time by the system. If the system required more than this number, an execution time error would be issued to the console terminal (if any).

9.3.3 Using a PTM Generated Clock

The real-time module (RTMOD), as supplied in the run-time library (FORLB.RO) permits use of either an externally generated 10-millisecond clock via a PIA, or the use of a PTM (MC6840-Programmable Timer Module) part in conjunction with the Phase 2 MPU clock.

If the PTM is used to generate the 10-millisecond clock interrupt, the initialization performed when SETRT is called programs the PTM properly. The initialization is based upon the use of a 1 MHz MPU clock. If another MPU clock frequency is used, the user must alter the initialization value. Timer #1 generates the interrupt in conjunction with the microprocessor (MPU) clock. In this configuration, pin 26 (G1) of the PTM (MC6840) must be grounded.

The value for initialization is found from the formula:

$$n = 0.01f - 57$$

where: n is the initializing value

f is the MPU clock frequency in Hz

Thus, for a 1 MHz (1,000,000 Hz) MPU clock:

$$n = (0.01)(1 \times 10^6) - 57$$

$$= 9943$$

A new value may be patched into the .LO file by relying on the map produced by RLOAD with the MAPF command. The named common PSCT name is ".PTMC" and consists of two bytes. Don't forget to convert the decimal number "n" into two bytes of hexadecimal for patching purposes with the MDOS PATCH command.

Alternatively, the user may assemble a short relocatable module and load after the library search (LIB=FORLB) has been done. Here is the source to be assembled:

```
          NAM    PTMVAL
          IDNT   PTM INITIALIZATION VALUE FOR X.X MHZ MPU
.PTMC     COMM   PSCT
          FDB    n      PUT CALCULATED DECIMAL VALUE OF n HERE
          END
```

Use RASM with the "R" option to produce the ".RO" file.

9.3.4 Using a PIA for Clock

When a PIA is used, the external 10 millisecond clock signal is brought into either CA1 or CA2 interrupt inputs. The PIA must be wired so that its interrupt output pin is connected to the IRQ input pin of the microprocessor.

9.4 TASK SUBPROGRAMS

A TASK subprogram is similar to a regular FORTRAN subroutine written to handle interrupts. However, a TASK cannot be CALLed like a SUBROUTINE, it can only be invoked by placing in the active queue by either a START (or STARTV) or using ATTACH and receiving an interrupt.

Except for the use of COMMON, only one byte of data can be passed to a TASK, and then only through the queue.

GENERAL FORM:

TASK <name>

or

TASK <name>(p)

where: <name> specifies the name of the task.

p specifies an optional parameter.

This statement must be the first statement in a task subprogram, and is similar to a SUBROUTINE statement. Each task thus defined acts as an independent program, and may include any valid FORTRAN statement except PROGRAM, SUBROUTINE, or FUNCTION. It may have one or more RETURN statements, or none at all. It must end with an END statement at the physical end of its source statements.

Any TASK may invoke other tasks, call upon subroutines or functions, and use COMMON in the normal manner.

9.5 START SUBROUTINE

The call to START is one of the methods of invoking a task.

GENERAL FORM: CALL START(<name>,i,j,k)

Where: <name> specifies the task name previously declared EXTERNAL.

i specifies the minimum amount of time to delay before executing the specified task. Negative or zero values indicate no delay is required prior to execution.

j specifies the value of the time unit associated with argument i as follows:

0 - unit of real-time clock (10 ms.)

1 - 10 milliseconds

2 - seconds

3 - minutes

k specifies a return argument which indicates if the task was accepted, where:

1 - specifies accepted

2 or greater - specifies not accepted

The execution of this call has the effect of queuing the invoked task with the same priority as the current task. Control is maintained by the current task.

9.6 STARTV SUBROUTINE

A slight variation of the START subroutine permits a single argument to be passed to the invoked task as well as allowing the task to be invoked with a different priority from the currently executing task.

GENERAL FORM: CALL STARTV(<name>,i,j,k,arg,pri)

where: <name> specifies the task name previously declared EXTERNAL.

i specifies the minimum amount of time to delay before executing the specified task. Negative or zero values indicate no delay is required prior to execution.

j specifies the value of the time unit associated with argument i as follows:

0 - unit of real-time clock (10 ms.)

1 - 10 milliseconds

2 - seconds

3 - minutes

k specifies a return argument which indicates if the task was accepted, where:

1 - specifies accepted

2 or greater - specifies not accepted

arg specifies the task argument (integer only).

pri specifies the priority level of the invoked task.

The execution of this call has the effect of queuing the invoked task with the specified priority level. Control is maintained by the current task.

9.7 ATTACH SUBROUTINE

This subroutine sets up an association between an interrupt and the task which is to handle it.

GENERAL FORM: CALL ATTACH(<name>,addr,nnn,pri)

where: <name> specifies the task to be associated with the interrupt, the name previously declared EXTERNAL.

addr specifies the memory address of the peripheral device.

nnn specifies the interrupt mask bit of the peripheral device in cases of a PIA or PTM, or the driver address for other devices handled through drivers. RESTRICTION: Driver address must be above 00FF hexadecimal. See Chapter 10.

pri specifies the priority level of the task.

With this subroutine, a given task can be assigned to handle multiple peripheral devices. The real-time operating system will prevent the same task from executing simultaneously for two or more interrupts. The subroutine makes an entry in the interrupt queue upon execution. The interrupt queue is searched when an interrupt is received.

9.8 WAIT SUBROUTINE

This subroutine enables the currently executing task to be suspended from execution for a period of time, allowing other tasks to be executed.

GENERAL FORM: CALL WAIT(i,j,k)

where: i specifies the minimum amount of time to delay before resuming this task. Negative or zero values indicate no delay is required prior to resumption.

j specifies the value of the time unit associated with argument i as follows:

0 - unit of real-time clock (10 ms.)

1 - 10 milliseconds

2 - seconds

3 - minutes

k specifies a return argument which indicates if the task was accepted, where:

1 - specifies accepted

2 or greater - specifies not accepted

9.9 WAITE SUBROUTINE

This subroutine suspends the currently executing task until a given event happens. Control will not be returned to the task until the variable has a value of zero. This suspension allows other tasks to be executed in the meantime.

GENERAL FORM: CALL WAITE(arg)

where: arg specifies an integer variable with a value from 0 to 255. (Only the least significant byte is used.)

NOTE

The variable specified is set to 1 after resumption to the task.

9.10 OTHER REAL-TIME SUPPORT SUBROUTINES

Some other subroutines which the real-time programmer may find useful are described in this section.

9.10.1 QCLEAR

This subroutine facilitates possible execution time error recovery if the data pool array containing the queues becomes overloaded. Execution of the subroutine essentially clears all real-time queues, enabling the programmer to start over. No arguments are permitted.

9.10.2 Single Byte I/O

Two subroutines enable the programmer to directly read and write single bytes to or from memory. This allows initialization of peripherial devices and has many other uses to the advanced programmer who must manipulate data in memory at the byte level.

GENERAL FORMS:
```
CALL BI(adr,var)
CALL BO(adr,arg)
```

where: adr specifies the memory address.

var specifies the variable to receive the data.

arg specifies a constant or variable containing data.

The subroutine BI is used to input one byte (Byte In), while BO is used to output one byte (Byte Out). All variables and constants must be integer.

9.10.3 Double Byte I/O

Two subroutines enable the programmer to directly read and write double bytes to or from memory.

GENERAL FORMS:
```
CALL DBI(adr,var)
CALL DBO(adr,arg)
```

where: adr specifies the memory address

var specifies the variable to receive the data

arg specifies a constant or variable containing data

The subroutine DBI is used to input two bytes (Double Byte In), while DBO is used to output two bytes (Double Byte Out). All variables and constants must be integer.

9.10.4 Bit Manipulation

The function of bit manipulation operations is to provide an efficient means of packing and testing data. This is particularly useful in microprocessor programs, especially in relation to I/O handling. These routines are implemented as functions.

Function Name	Operation
IOR(j,m)	Performs logical inclusive OR.
IAND(j,m)	Performs logical AND (product).
INOT(j)	Performs logical complement.
IEOR(j,m)	Performs logical exclusive OR.
ISHFT(j,m)	Performs logical shift.
IBTEST(j,m)	Performs test of specific bit.
IBSET(j,m)	Performs setting of specific bit.
IBCLR(j,m)	Performs clearing of specific bit.

These and other functions are described in more detail in Appendix D.

9.11 REAL-TIME PROGRAMMING HINTS

The methods used and the philosphy behind real-time systems are so different from conventional FORTRAN programming that it is appropriate to cover certain essential points. This is especially necessary because of timing considerations and the interactions between the programmer's code and the operating system.

9.11.1 Use of the RETURN Statement

At least one RETURN (or END) statement must be used in every SUBROUTINE, FUNCTION, or TASK. This statement may also be used in a main real-time program if the main program does not contain instructions to be executed after execution of the initialization tasks.

Usually the beginning of the main program will perform task initialization. After initialization, there may be instructions to be executed in background or there may not. If there is, then the background code will consist of an endless loop. If there is no background code, then the RETURN statement is used. The reason for this non-standard usage is that any main program of a real-time system is itself considered to be a subroutine of the operating system.

9.11.2 Multiple Interrupts

External interrupts are handled by the TASK subprogram feature. Such interrupts may occur at any time and on occasion may follow one another very rapidly. It may, therefore, happen that while one external interrupt is being handled by its associated task, another external interrupt may occur. If this happens, the second interrupt will be placed in the active queue by the operating system.

If the executing task contains a call to WAIT, WAITE, or an I/O statement, this will cause the operating system to suspend its operation and return to processing the active queue.

The operating system will not permit the task to be re-entered until it has completed execution on behalf of the first interrupt. This lockout provision thus prevents execution confusion and allows several such interrupts to be queued for execution in an orderly manner.

9.11.3 Data Read at Interrupt

When an external interrupt has been sensed, data will always be read from the corresponding PIA register or handled by the device driver. Note that this will occur at time of the interrupt, not at the time at which the task associated with the interrupt is executed. This data is available to the task if an argument was used in the TASK statement.

9.11.4 Task Sharing Same Subroutines

Two tasks may call the same standard subroutine if that subroutine does not contain a call to WAIT, WAITE, or an I/O statement. In order to understand the reason for this, consider the following example:

```
TASK A                          TASK B
 .                               .
 .                               .
CALL XYZ                        CALL XYZ
 .                               .
 .                               .
RETURN                          RETURN
END                             END
```

```
SUBROUTINE XYZ
  .
  .
CALL WAIT(2,2,K)
  .
  .
RETURN
END
```

Task A is entered first and call subroutine XYZ. XYZ is then executed until the 2 second delay is encountered and control returns to the operating system. If another interrupt is now sensed which starts task B, subroutine XYZ will again be called and the system will fail.

A little consideration will show that this situation cannot occur if there is no WAIT, WAITE, or I/O within subroutine XYZ, since there will then be no possibility of returning to the operating system.

9.11.5 Processing Necessary Responses

The manner in which the operating system works, encourages the user to design interrupt tasks and subroutines which are relatively short. Lengthy computational routines may cause the system to be locked out of processing necessary responses to interrupts or other high frequency routines. A simple method of subdividing long computational routines is to use the statement: CALL WAIT(0,0,K) at appropriate intervals. This has the effect of passing control back to the operating system. To this end, a subroutine named WAITZ is supplied and eliminates the necessity and overhead of the (0,0,K) arguments.

9.11.6 Task Stack Size Limitations

Each TASK subprogram has memory allocated for a stack area. The stack must handle return addresses for subroutine and function calls and at least 7 bytes for each interrupt received. A compromise size of 32 bytes is allocated by the compiler in DSCT. If the user finds this size is too small, he may easily increase this size by loading a simple module before loading each TASK module in which he desires the stack size allocation increased.

The assembly language source for this allocation module is shown below. It must be assembled as a relocatable module by the MDOS Macro Assembler (RASM). It may be repeatedly loaded as needed.

```
NAM INCSTK
IDNT INCREASE DSCT STACK ALLOCATION BY $20
DSCT
RMB $20 (This value may be changed as desired)
END
```

9.12 END-SYSTEM HARDWARE CONSIDERATIONS

9.12.1 Real-Time Clock

A real-time clock is necessary for the operation of the REAL-TIME FORTRAN operating system. It has been determined that a 10 millisecond clock is a reasonable compromise in a microprocessor system between response time to interrupts, accuracy of timing, and overhead time associated with each "tick" of the clock.

Two methods are provided to implement this clock in hardware. One is to use a clock oscillator and bring it into the system via a PIA. The other is to use a PTM and generate the clock signal as a function of the MPU clock signal. MDOS REAL-TIME FORTRAN makes the assumption that if the PTM is used, the MPU clock frequency is 1.0 MHz. If other than this frequency, the routine in SETRT must be patched to take this into consideration.

9.12.2 No Console in System

A system can be devised that does not have any kind of operator's terminal or console I/O device. If this is the desired system, then several items must be considered.

First, what will happen if an execution time error occurs? There should be some provision to notify the operator of the end-system of this fact.

Second, it will be necessary to put a "dummy" address of 0000 for the ACIA in the CALL SETRT statement. This causes the operating system to ignore any initialization or attempt at I/O via the ACIA.

9.12.3 MDOS Disk I/O

MDOS Disk I/O operation in a Real-Time system will cause the real-time clock interrupts (ticks) to be ignored during accesses to the disk. Therefore, any waits or other timing dependent upon the real-time clock would no longer be accurate or necessarily consistent. When MDOS disk I/O is used, the PSCT portion of the program cannot be put into any type of ROM.

9.13 VECTORS FOR NMI, IRQ, AND RESTART

The following externally defined symbols may be used in defining the upper ROM vectors for the end-system:

```
IRQ$      IRQ vector (at $FFF8)
NMI$      NMI vector (at $FFFC)
START$    RES vector (at $FFFE)
```

The NMI in this Real-Time operating system saves the stack pointer in memory, sets a flag in another location, and then just goes into an endless loop.

The system may be restarted from the NMI condition by re-entering at the location defined by the externally defined symbol "RESTR$". This entry point checks for the NMI flag and, if it finds it present, will reload the stack pointer from the location it was saved in previously. If it does not find the NMI flag, the stack is loaded from a predetermined location - the same as if the "START$" entry point was used. Therefore, "RESTR$" could be used in place of "START$" for the RES vector at $FFFE.

The NMI flag pattern used is $C3A5 and relies upon RAM memory not initializing to that exact two-byte pattern upon a power-up.

9.14 DEBUG OF REAL-TIME PROGRAMS

To assist the programmer in debugging Real-Time FORTRAN programs, certain tools have been developed. This section gives certain information necessary to be able to find the cause of a system crash or other malfunction.

It has been found to be almost a necessity to be able to disable temporarily the real-time clock interrupts from the system while certain debug procedures are used.

9.14.1 Queue Entry Formats

The following information is presented here to aid an advanced real-time programmer in determining malfunction causes.

QUEUE ENTRIES:

ACTIVE QUEUE (External reference link: AQ$)

Bytes	Used for
0-1	Link to next entry
2	Priority level
3	Stack Flag: 0=Entry mode/new stack, 1=Use old stack
4-5	Re-entry (task) or stack address
6-7	Data
8-9	Lock cell address

TIMER CONTROL (External reference link: TQ$)

Bytes	Used for
(10 ms. queue)	
0-1	Link to first entry in queue array data pool
2	Initial value (1)
3	10 ms. Counter
(1 sec. queue)	
4-5	Link to first entry in queue array data pool
6	Initial value (100)
7	1 second counter
(1 min. queue)	
8-9	Link to first entry in queue array data pool
10	Initial value (60)
11	1 minute counter

TIMER QUEUE ENTRY FORMAT (first entry found from TIMER CONTROL)

Bytes	Used for
0-1	Link to next entry
2	Priority level
3	Stack Flag: 0=Entry mode/new stack, 1=Use old stack
4-5	Re-entry or stack address
6-7	Data
8-9	Counter value

INTERRUPT QUEUE (External reference: IQ$)

Bytes	Used for
0-1	Link to next entry
2	Priority level
3	Bit to test (mask)/00=driver
4-5	Re-entry (task) address
6-7	Driver address or 0000
8-9	PIA/PTM address/device address

9.14.2 QDUMP Subroutine

This subroutine is supplied as part of the REAL-TIME FORTRAN run time library (FORLB.RO). It may be called at any time during a program and will use whatever FORTRAN console I/O is provided by the user. The purpose of the subroutine is to produce a dump of the various queues, in an orderly fashion, to either the console or the line printer (the user is queried each time it is called).

A possible use of this subroutine might be to call it whenever any error or a particular error is observed. Error calls may be intercepted through the module named ERROR.

9.14.3 Active Queue Dispatch Logging

A collection of several subroutines enable logging of all dispatches from the active queue. These subroutines are:

```
SETRTD
RTDON
RTDOFF
RTDDMP
```

The feature is invoked by calling SETRTD and supplying the name of a two-dimensional integer array to hold the data and the mode of operation. See Appendix E for a further description. The dimension statement should be as follows:

```
DIMENSION  LOGARY(6,i)
```

where: i is the maximum number of data entries.

The logging is started by calling RTDON and halted by RTDOFF. The data may be printed by RTDDMP.

The data logged includes the real-time clock tick counter (TIC$), data bytes 2-9 of the Active Queue, and the stack pointer value. A header is printed to identify the six columns of hexadecimal data.

CHAPTER 10

EXTERNAL DEVICE DRIVERS

10.1 INTRODUCTION

MDOS FORTRAN supports external device drivers in a way which makes I/O to devices other than console terminal, line printer, and MDOS disk quite easy and efficient.

Most device drivers will be written in assembly language and assembled as a relocatable module to be linked to FORTRAN when required. Normal FORTRAN statements such as OPEN, CLOSE, READ, and WRITE will be used by the programmer to access most any device.

10.2 FORTRAN I/O STATEMENTS

The MDOS FORTRAN statements which may be used by the programmer in implementing external devices for I/O are described in this context in the following paragraphs.

10.2.1 EXTERNAL

The EXTERNAL statement is used to declare the name given to the device driver as external to the program, distinguishing it from an internal variable or subprogram name.

EXAMPLE: EXTERNAL ACIADV

where: "ACIADV" is the name of the driver routine to be used.

10.2.2 OPEN

Once the driver routine has been declared EXTERNAL, it may be associated with a particular FORTRAN file reference number with an OPEN statement. This is very similar to opening a disk file, except that the external name of the driver routine is supplied in place of the disk file name, and the external device address is supplied in place of the file mode.

GENERAL FORM: OPEN (n,x,a)

where: n specifies the file reference number

x specifies the external device driver routine name

a specifies the external device address

The same driver may be used for more than one device by additional OPEN statements referencing that driver.

EXAMPLE: OPEN (7,ACIADV,$EC90)

This example causes the following action upon execution:

1. Associates unit number 7 with using a driver name "ACIADV" for a device physically located at memory address $EC90.

2. Goes to the driver routine "ACIADV" to perform any initialization of the device located at $EC90. For some devices, no initialization is necessary. See paragraph 10.4.1.

NOTE

Use of the OPEN statement in conjunction with an EXTERNAL routine overrides any previous assignment of a file reference number to either a disk file or a pre-assigned unit (such as 102 for line printer) until a corresponding CLOSE statement is issued, at which time any previous assignment is restored.

10.2.3 READ/WRITE

After the external driver routine has been defined and a reference number assigned, normal READ and WRITE statements may be used in conjunction with FORMAT statements to perform device I/O through the driver routine.

```
EXAMPLE:        WRITE (7,900)
          900   FORMAT (' ENTER NUMBER & NAME >> ')
                READ (7,901) INUMB,NAME1
          901   FORMAT (I3,7A2)
```

Notice that normal READ, WRITE, and FORMAT statements are used. This is true except where only certain bits or bytes are required to be changed on output. See the "B" format editing code for an explanation on how to do this.

10.2.4 CLOSE

Termination of the external device is done by a CLOSE statement.

EXAMPLE: CLOSE (7)

This statement causes the following action upon execution:

1. The driver executes any desired termination routine for the device.

2. Unit 7 is dis-associated from the driver and device and is now available for re-use if desired.

10.3 SUPPORTING SUBROUTINES

There are two supporting subroutines available for use with an external driver. These are DEVON and DEVOFF.

During execution of a FORTRAN program, additional ON/OFF control may be optionally exercised in relation to the external device. Depending upon how this feature is implemented in the device driver, a CALL DEVON or CALL DEVOFF will take the intended action.

GENERAL FORM: CALL DEVON(n)
CALL DEVOFF(n)

where: n specifies the file reference number assigned by an OPEN statement.

10.4 DRIVER STRUCTURE

All external device drivers used with MDOS FORTRAN must adhere to certain conventions. These are outlined in the following paragraphs.

10.4.1 VECTOR TABLE

Each driver must have a vector table, the start of which corresponds to the XDEF of the driver name. The vector entries are described below:

Bytes	Pointer to Function	Called by
0-1	Initialize the device	OPEN
2-3	Terminate the device	CLOSE
4-5	Input to I/O buffer	READ
6-7	Output from I/O buffer	WRITE
8-9	Poll for IRQ	RTMOD routine (Real-Time version only)
A-B	Reserved	
C-D	Turn on device	DEVON
E-F	Turn off device	DEVOFF

If any particular function is not implemented, the vector address given should point to an RTS instruction. All vector routines must end with an RTS.

The device address (if any) is passed to the driver through an externally defined symbolic address of DV$ADR, except for IRQ handling where accumulators A and B are used. I/O is passed back and forth between FORTRAN and the driver through a buffer defined by the symbol BUF$.

On a WRITE statement in FORTRAN, one formatted line of output is placed in BUF$ buffer, then control is passed to the driver (through the vector at bytes 6-7). It is then the responsibility of the driver to take the data from the buffer and send it out to the external device.

On a READ statement in FORTRAN, control is passed to the driver (through the vector at bytes 4-5). It is the driver's responsibility to receive data from the external device, place it in the BUF$ buffer with an ASCII EOT ($04) character at the end, and then return control (via RTS) to FORTRAN to get the data from the buffer and place it in the variable list associated with the READ statement.

10.4.2 BUFFERS

Normally, most of the I/O will use only BUF$ as the buffer. However, in certain interrupt driven systems, it may be desirable for the device driver routine to have an additional buffer of its own. This allows the driver to transfer at high speed the contents of its own buffer to BUF$ or vice-versa, when needed, thus freeing up BUF$ for other I/O in the system.

An example of this might be when a system was writing to a line printer and inputting from the keyboard at the same time. Here, it would be advantageous for the keyboard input driver and line printer driver routines to each have their own buffer, using BUF$ only when needed by FORTRAN.

10.4.3 INTERRUPT HANDLING (Real-Time Only)

Since interrupts may come from many different sources in a system, software polling must be done to find the source of the interrupt. Provision has been made through driver vector bytes 8-9 to allow polling of the external device for an interrupt. Accumulators A and B will contain the device address (A most significant byte of address). The driver must return accumulator A cleared if the device did not cause the interrupt, or accumulator A as non-zero if an interrupt is detected. In addition, any data to be returned upon detecting an interrupt must be passed in the index register by the driver.

Clearing of the interrupt source is accomplished through this driver routine before return to the caller.

10.4.4 Driver Address Restrictions

If the subroutine ATTACH is used, a device driver cannot start at any address below $0100. Normally, this is no restriction to be concerned with as most systems will use this area for either RAM or I/O devices - not program memory.

10.5 SAMPLE DRIVERS

The following is a source listing of a general purpose ACIA driver, which may be modified by the user to suit the application. Interrupts are inhibited in this version. Assumption is made that the ACIA clock divide ratio is 16 and that 7 bits of data, 1 stop bit, and even parity is being used.

```
          NAM ACIADV
          XDEF ACIADV
          XREF BUF$,DV$ADR

          DSCT
BPTR      RMB 2                    BUFFER POINTER STORAGE

          PSCT
ACIADV    EQU *
          FDB DEVINT
          FDB DEVTRM
          FDB DEVIN
          FDB DEVOUT
          FDB DEVIP
          FDB DUMMY
          FDB DEVON
          FDB DEVOFF

* UNIMPLEMENTED VECTORS
DEVTRM    EQU *
DEVIP     EQU *
DUMMY     EQU *
```

```
DEVON  EQU *
DEVOFF EQU *
       RTS

* INITIALIZATION OF ACIA
DEVINT LDX DV$ADR              GET ACIA ADDRESS
       LDAA #3
       STAA 0,X                MASTER RESET
       LDAA #%00001001
       STAA 0,X                INITIALIZE
       RTS

* INPUT FROM ACIA
DEVIN  LDX #BUF$+1
DEVIN2 STX BPTR
       LDX DV$ADR              GET ACIA ADDRESS
DEVIN4 LDAA 0,X                STATUS
       LSRA
       BCC DEVIN4              NOT READY
       LDAA 1,X                GET DATA
       CMPA #$0D               CR?
       BEQ DEVIN9              YES
       LDX BPTR                GET BUFFER POINTER
       STAA 0,X
       INX
       CPX #BUF$+132           END OF BUFFER YET?
       BNE DEVIN2
DEVIN9 LDAA #4                 EOT
       STAA 0,X                MARK END
       RTS

* OUTPUT TO ACIA
DEVOUT LDX #BUF$+1
DEVO2  STX BPTR
       LDAA 0,X                GET CHARACTER
       CMPA #4                 EOT?
       BEQ DEVO9
       BSR SEND
       LDX BPTR
       INX
       BRA DEVO2

DEVIN9 LDAA #$0D               CR
       BSR SEND
       LDAA #$0A               LF
       BSR SEND
       CLRA                    NULL
       BSR SEND
       RTS

SEND   LDX DV$ADR              GET ACIA ADDRESS
SEND2  LDAB 0,X                STATUS
       LSRB
       LSRB
       BCC SEND2               NOT READY
       STAA 1,X                SEND CHARACTER
       RTS

       END
```

CHAPTER 11

INTERFACING WITH MICROMODULES

11.1 INTRODUCTION

Micromodules are a series of Motorola OEM boards, each with various features and functions. MDOS 6800 REAL-TIME FORTRAN has been written with these boards in mind and, in most cases, can be used in a system comprised of one or more Micromodules. This chapter describes the interfacing and use of some Micromodules with FORTRAN.

11.2 MICROMODULE 14/14A

Micromodule 14 (and 14A) allows use of an Arithmetic Processor Unit (APU) with 6800/6809 family microprocessors. The REAL-TIME version of MDOS FORTRAN may interface and use the APU on the Micromodule 14 and 14A boards to increase execution speed of real number arithmetic operations and allow several more trig functions than present in the FORTRAN library. The additional functions are:

ASIN
ACOS
ALOG10

11.2.1 Using MM14 or MM14A

The programmer simply specifies use of a Micromodule 14 (or 14A) board in the end-system at compile time through the "M" option letter on the command line. See Chapter 1 concerning the command line of the compiler.

The rest is automatic, as during link time, the proper modules from the FORTRAN library will be searched for and loaded.

11.2.2 MM14/14A Precautions

The programmer must assure that ALL modules (outside the supplied FORTRAN library) have been compiled with the "M" option; otherwise, there will be a symbol conflict during link time.

The FORTRAN program makes no checks to ascertain that an MM14 board is present in the system. If the board is not present in the system at execution time, rather unpredictable results will take place. The program probably will either hang up in a loop or will abort due to an execution time error (usually an overflow error).

11.2.3 Relocating MM14/14A Base Address

MM14/14A (Micromodule 14/14A) is supplied with the base address wired for memory address $EC30. It can be changed on the board by the user to a different address through the following directions in the MM14 manual.

If the base address is changed, the programmer must convey the new address to the linking loader (RLOAD). This may be done by either of two methods:

a. Supply the definition of the symbol MM14$ to RLOAD by the commnd:

```
DEF:MM14$=$nnnn
```

where: nnnn is the new base address.

This must be done BEFORE the library search is done (LIB=FORLB).

b. Assemble a relocatable module containing an XDEF to MM14$ and define the new base address. This module must be loaded before the library search. The module could be as follows:

```
        NAM     MM14XX
        XDEF    MM14$
        IDNT    MM14 BASE ADDRESS DEF $C000
MM14$   EQU     $C000  NEW BASE ADDRESS
        END
```

11.3 MICROMODULE 12/12A

Certain rules apply to interfaces with MM12 and MM12A. Generally, as long as a programmer is aware of these rules, they should not inhibit the usefulness of the micromodules or the interfaces.

1. For a particular device on the GPIB, its talk address and listen address must be the same. This is normally the only way it can be done since usually only one address switch is provided.

2. The GPIB address of all devices must be in the range of 1 to 30, inclusive. Addresses outside this range are not allowed.

3. The listen/talk addresses for MM12 or MM12A cannot have a secondary address. Other devices on the GPIB may have a secondary address.

4. Only one MM12 or MM12A module may be interfaced with MDOS FORTRAN on a microcomputer. This does not preclude several microcomputers in a total system, each with its own MM12 or MM12A module.

11.3.1 MM12 - GPIB Listener/Talker/Controller Module

Interface with this module and FORTRAN is accomplished by use of a set of FORTRAN callable subroutines, drivers, and special FORTRAN statements. These make use of the firmware supplied as part of the MM12 module. Useful subroutines, FORTRAN statements, and syntax are shown below.

EXTERNAL MM12

Declares the driver name external so it will not be confused with local variable names. MM12 is the name of the supplied driver in the library.

OPEN (frn,MM12,ba)

where: frn — is the FORTRAN I/O reference number, which is the GPIB address of MM12 module talker/listener. It must be an integer value (constant or variable) of 1 to 30. The MM12 cannot have a secondary address.

MM12 — is the name of the driver.

ba — is the base address of the MM12 firmware (normally at $B800 unless altered by user). This value is not used or checked, but is included for documentation and consistency.

This statement provides the power-on initialization of the MM12, defines the MM12 bus address, and associates the FORTRAN I/O reference number with the driver for later use.

READG (ta,la,fmt) list

where: ta — is talker bus address.

la — is listener bus address (may be integer or an integer array containing 1 or more listener addresses). The address of the MM12 must be designated as a listener.

fmt — is the format statement number.

list — is the list of one or more variables to receive the data read.

WRITEG (ta,la,fmt) list

where: ta — is talker bus address. The address of MM12 must be designated as the talker.

la — is listener bus address (may be integer or an integer array containing 1 or more listener addresses).

fmt — is the format statement number.

list — is an optional list of variables whose values will be output.

CALL ATTACH(name,device,MM12,pri)

where: name — is the task name, previously declared EXTERNAL.

device — is the GPIB address of the device.

MM12 — is the name of the driver.

pri — is the priority level for the task.

This call sets up service request interrupts for MM12 if desired. NOTE: MM12 board must have jumper added from 11 to 12 of K1 to enable an SRQ interrupt. ENSRQI must be called to enable the interrupt.

CLOSE (frn)

where: frn — is the FORTRAN I/O reference number.

This statement releases the reference number and masks PIA #1 of MM12 so that an interrupt will no longer be recognized through the ATTACH previously used.

In addition to the above routines, other individual functions may be handled through the other supplied library routines as follows:

CALL MRST — Performs a master reset of all devices on the bus. The state of the instruments are reset to the conditions specified by the manufacturer.

CALL ENFP(addr) — Enables Front Panel controls of the instrument(s) specified by the bus address(es) (addr).

CALL LLO(addr) — Locks out the front panel controls of the instrument(s) specified.

CALL TSETUP(ta,la) — (TALK SET-UP)

where: ta — is the bus address for the talker

la — is the bus address for the listener(s).

This subroutine sets up the bus for the designated talker and the one or more designated listeners. One talker and at least one listener must be specified.

CALL LISTEN(addr) — Sends the listen address(es) specified.

CALL UNL — Sends the unlisten command.

CALL UNT — Sends the untalk command.

CALL UNTUNL — Sends both the untalk and unlisten commands.

CALL TALK(addr) — Sends the talk address specified.

CALL TSTSRQ(addr,code)	If the specified device (addr) generated a service request, returns code=1, otherwise returns code=0. A code of -1 will be returned if a parallel poll were conducted and the device address was not previously declared with a call to PPR.
CALL PPR(addr,line)	Parallel poll response. Enters the line position (1-8) and device address in a parallel poll table.
CALL POLTYP(code)	Determines if a request for service test (GSRQ) is to be accomplished using a serial or parallel poll. The mode is specified by using code=0 for serial (default) or code=1 for parallel.
CALL RQS(status)	Sends a request for service to the active controller.
CALL ENSRQI	Enables IRQ to be generated by MM12 with SRQ.
CALL PASCTL(addr)	Passes bus control to the controller specified.
CALL RESETG(addr)	Resets only the device(s) specified.
CALL GETRIG(addr)	Group Execute Trigger. Triggers the device(s) specified.
CALL WT4CTL	Waits until the active controller passes control to this controller (MM12).
CALL SETEOT(arg)	Sets EOT byte to value specified in the argument. If two non-zero bytes are specified, these bytes will be sent as termination characters by the WRITEG routine. Default is $0D0A (CR,LF). May be changed as often as needed by this call. The least significant byte (default $0A) will be interpreted as EOT (End of Transmission) character by READG.

11.3.1.1 Compiler Option G

Due to the nature of handshaking on the GPIB with MM12, it is necessary to periodically enter the firmware on MM12 to assure completion of handshaking when the MM12 is not the controller-in-charge (CIC).

When operated in a real-time system, this will be done every clock interrupt. However, in a non-real-time environment when MM12 is not the CIC, the GPIB will hang up whenever a command byte is sent on the bus, unless the MM12 firmware is entered to complete the handshake.

Therefore, this version of FORTRAN incorporates two methods of accessing the MM12 firmware:

1. Whenever console or line printer I/O is being done, the MM12 firmware will be accessed if:

    a. there is an MM12 in the system, and

    b. the MM12 is not the CIC, and

    c. it is waiting for the console I/O ACIA to become ready, or on every character output to the line printer.

2. The programmer may selectively turn on and off the "G" option. When turned on, the G option will produce JSR (Jump to Subroutine) code after every FORTRAN source statement. The subroutine called (ET$R30) checks for MM12 being the CIC and, if not, enters the firmware. Note that this produces an overhead of three bytes per source line and some time delay when invoked.

The G option may be turned on by "$G", with the "$" in column 1. Alternatively, it may be turned off with "$-G". It need only be used during the portion of a program executing when the MM12 is not the CIC.

If the GPIB can withstand the delay, the G option does not have to be used. Likewise, it should not be used in a real-time FORTRAN PROGRAM.

11.3.1.2 Relocating MM12 Base Address

MM12 is supplied with on-board firmware starting at address $B800. If it is desired to change this address, the user must reassemble the firmware, reprogram the EROM devices and, in addition, must alter the hardware connections as per the MM12 manual.

If the base address is changed, the programmer must convey the new address to the linking loader (RLOAD). This may be done by either of two methods:

a. Supply the definition of the symbol MM12$ to RLOAD by the command:

```
DEF:MM12$=$nnnn
```

where: nnnn is the new base address.

This must be done BEFORE the library search is done (LIB=FORLB).

b. Assemble a relocatable module containing an XDEF to MM12$ and define the new base address. This module must be loaded before the library search. The module could be as follows:

```
        NAM     MM12XX
        XDEF    MM12$
        IDNT    MM12 BASE ADDRESS DEF $8C00
MM12$   EQU     $8C00  NEW BASE ADDRESS
        END
```

11.3.2 MM12A - GPIB Listener/Talker Module

The use of a software driver module (MM12A) and several special subroutines allow easy interface of this Micromodule to MDOS REAL-TIME FORTRAN. The following lists the various interfaces (more detailed information on the FORTRAN statements may be found in Chapter 10).

EXTERNAL MM12A

Declares the driver name external so it will not be confused with local variable names. MM12A is the name of the supplied driver in the library.

OPEN (frn,MM12A,ba)

where: frn — is the FORTRAN I/O reference number, which must be the same as the GPIB address of the MM12A module. It must be an integer value (constant or variable) of 1 to 30. The MM12A cannot have a secondary address.

MM12A — is the name of the driver.

ba — is the base address of the GPIA device on the MM12A module.

This statement associates the address and drivers and performs initialization of the GPIA.

READ (frn,fmt) list

where: frn — is the FORTRAN I/O reference number.

fmt — is the FORMAT statement number.

list — is the list of one or more variables to receive the data read.

This statement reads data through the MM12A Listener from the GPIB.

WRITE (frn,fmt) list

where: frn — is the FORTRAN I/O reference number.

fmt — is the FORMAT statement number.

list — is an optional list of variables whose values will be output.

This statement writes data through the MM12A talker to the GPIB.

CLOSE (frn)

where: frn is the FORTRAN I/O reference number.

This statement releases the "frn", thus disassociating the addresses and drivers. It also resets the GPIA.

CALL SETEOT (arg) Sets EOT byte to value specified in the argument. If two (2) non-zero bytes are specified, these bytes will be sent as termination characters by the WRITE routine. Default is $0D0A (CR,LF). May be changed as often as needed by this call. The least significant byte will be considered the EOT by READ.

CALL RQS12A(status) Sends a request for service to the active controller.

CALL CRQS Clears the request for service.

CALL LPE(status) Sets up the parallel poll response with the status byte.

11.4 MM15A, MM15A1 - A/D 8, 16, or 32 channel

A driver is supplied for these micromodules. The usual EXTERNAL declaration and OPEN statements are used. However, to "read" a channel, the user will CALL a library supplied subroutine. Actually, two of these "read" subroutines are supplied - one for data in 2's complement; the other for data in unsigned binary.

The driver is named: MM15A. Therefore, the following would be an example of use in a FORTRAN program:

```
EXTERNAL MM15A

OPEN (frn,MM15A,addr)
```

where: frn is the FORTRAN I/O reference.

addr is the base address of the MM15A or MM15A1.

The syntax for the subroutine call is:

CALL R15AS(frn,cn,gain,ivar) for signed results.

CALL R15AU(frn,cn,gain,ivar) for unsigned results.

where: frn is the FORTRAN I/O reference number.

cn is the channel number.

gain is the gain (1, 2, 4, or 8).

ivar is the variable in which the result will be returned.

The result returned will be in integer form acceptable to FORTRAN for arithmetic calculations. It will not take into account the gain used nor the range selected (through hardware strapping).

Since the conversion time is 40 microseconds maximum, no use of the interrupt feature of this module is made because the overhead associated with interrupts in Real Time FORTRAN is greater than any possible time savings. The processor is allowed to run in a loop, waiting for the end of conversion.

No over- or under-range error condition is returned by this module.

In addition to the above, there is available another set of "read" subroutines (called R15ASA and R15AUA) which features auto-ranging. Operation from a FORTRAN viewpoint is the same except that the read routines return both the value and the range actually used. These routines may perform more than one actual channel read conversion to obtain a result with the most significant digits possible without overflow. The first read attempt will use the "gain" supplied in the calling argument. Caution should be observed to use a variable (not a constant) in the gain parameter of the call.

11.5 MM05A, MM05B - A/D 8 or 16 channel

These modules are handled quite like the MM15A module. The driver is MM05A or MM05B. The "read" subroutines use the following syntax:

CALL R05A(frn,cn,ivar) for both signed and unsigned results on MM05A.

CALL R05B(frn,cn,ivar) for both signed and unsigned results on MM05B.

where: frn is the FORTRAN I/O reference number,

cn is the channel number

ivar is the variable in which the result will be returned.

Since this module will halt the microprocessor during conversion, no use of the interrupts are made. Also, it should be noted that in systems using the PTM for the real-time clock generation, the clock "ticks" will be elongated if the "tick" occurs during the halting of the MPU. This probably will not be of any consequence for most systems; however, for systems requiring great accuracy in timing, the user may wish to provide an externally generated real-time clock (10 ms. repetition rate) instead of the PTM.

The usual OPEN, EXTERNAL, and CLOSE statements apply to these modules.

11.6 MM15CV, MM15CI - D/A 1 to 4 channels

A driver and a "write" subroutine are supplied for these modules. The driver is named MM15C and the output subroutine has the following syntax:

CALL W15C(frn,cn,ivar)

where:	frn	is the FORTRAN I/O reference number,
	cn	is the channel number (1-4)
	ivar	is the variable (or constant) containing the value to be output.

The value to be output may be signed or unsigned. The actual range is determined by hardware strapping. The usual EXTERNAL, OPEN, and CLOSE statements apply.

The module is initialized to 0 volts (4 ma. for CI module) or most negative voltage (for bipolar output) with the OPEN statement. The driver will attempt to initialize four channels, even though less may be present on the module. Therefore, base address selection should be made to allow for unused addresses (put the next module 8 bytes higher to allow for unused channels).

11.7 MM05C - D/A 4 channel

The driver is MM05C and the write subroutine is W05C with the same syntax as for MM15C series above.

11.8 MM15B - A/D 1 to 16 channels (with MM15BEX)

Since conversion time for this module is relatively long (133 Milliseconds max.), the read routine calls WAITZ until end of conversion is indicated. After the "read" subroutine is called, the conversion is started and control is returned to the real-time executive. Control will eventually be returned to the statement following the original "read" call.

Driver name is MM15B and the "read" subroutine has the following syntax:

CALL R15B(frn,cn,ivar)

where:	frn	is the FORTRAN I/O reference number,
	cn	is the channel number,
	ivar	is the variable in which the result will be returned.

The usual EXTERNAL, OPEN, and CLOSE statements are used with this module.

11.9 MM03, MM13A, MM13B, MM13C, MM13D

This series of Micromodule uses driver MM03 in the FORTRAN library. The MM03 module has both input and output capability, while MM13A and MM13B have only output capability, and MM13C and MM13D have only input capability.

Input (FORTRAN "READ" operation) is relatively simple and straightforward. Output to those modules (MM13A, MM13B, and MM03) is done with the FORTRAN "WRITE" statement in conjunction with a FORMAT statement using the "B" format edit code. This allows changing only the individual outputs desired, leaving the remainder unaffected.

As an example, let us suppose we are using an MM03 which has its 32 outputs set to this hexadecimal byte pattern: 52 40 37 08.

If it were desired to clear bits 0 and 4 and set bit 7 of the third byte, the proper statements could be:

```
        OPEN (72,MM03,$9FFC)
        KKKK=$80                ;data
        WRITE(72,901) KKKK
    901 FORMAT(2X,B$91)
```

FORTRAN will place the following bytes into the I/O buffer (BUF$):

```
        20 20 00 91 80 04
```

and the driver (MM03) will interpret them as skipping the first two-byte locations of the micromodule, and using hexadecimal 91 as the bit mask for the data 80 for the third location. The 00 byte is a flag for the driver, and the 04 is the ending character.

The net result will be the third byte changing to A6, since only the bits with a corresponding 1 in the bit mask will be changed.

OLD DATA	00110111	($37)
NEW DATA	10000000	($80)
BIT MASK	10010001	($91)
RESULT	10100110	($A6)

Note that it is not possible to read the old output condition in these micromodules to perform manipulation via software.

Several other features of the MM03 driver should be noted. First, it is possible to use the same driver with up to six different micromodules. The OPEN statement initializes all outputs to a data zero (OFF) pattern (outputs high in MM03, open contacts on MM13A and MM13B). The DEVON and DEVOFF calls will turn <u>all</u> outputs on or off. The CLOSE statement turns all outputs off (same as OPEN).

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)	*
+	, (comma)
-	(
/	' (apostrophe)
=	&
)	$
;	. (period)

Except in literal data, where any valid ASCII character is acceptable, the 50 characters listed above constitute the set of characters acceptable by MDOS FORTRAN.

APPENDIX B

COMPILER ERROR MESSAGES

When errors are detected by the compiler, the following message is printed at the console terminal:

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

where: "code" represents one of the coded errors in the list below. An asterisk will be printed on the line preceding the error code to indicate the scanning position when the error was detected.

EXAMPLE:

```
            IF(J-3 10,20,30
                 *
*** ERROR 05
```

00 illegal character

01 non-numeric statement number

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

06 too many arguments (13 maximum)

07 numeric value too large

10 duplicate statement label

11 name already defined

12 array dimension too large

13 COMMON variables cannot be initialized in DATA statements

14 name too long (6 character maximum)

15 PROGRAM, SUBROUTINE, TASK, or FUNCTION statement not first

16 DATA variable does not match data type

17 subroutine name and variable name conflict

18 must be integer argument

19 name not yet declared EXTERNAL

20 too many statement labels with computed GOTO (20 maximum)

22 dummy argument name already used

23 too many external references

24 common or dummy argument not permitted

25 EQUIVALENCE not permitted

26 E and F editing codes not permitted with I option

30 over 10 operands in this statement

31 number of subscripts does not agree with number of dimensions

50 too many nested DO's (10 maximum)

51 one of the DO arguments is not an integer

52 DO improperly terminated

53 END IF without matching IF-THEN

54 END IF missing

55 too many nested IF-THEN's (10 maximum)

56 branch out of range in logical IF

APPENDIX C

EXECUTION TIME ERROR MESSAGES

If a fatal error is detected during execution, the following message will be displayed on the console terminal:

```
***EXECUTION TIME ERROR #nn
    xxxx
    xxxx
    xxxx
    xxxx
```

In the above example, nn represents the error number, and the four lines of xxxx represent the last four double bytes found on the stack (SP on 6800 and S on 6809). These values normally represent the subroutine return addresses and can be of some aid in locating what routines were called/executing when the error was encountered.

The following is a listing of the execution time error numbers and their meanings:

01 POWER function cannot be called with -X

02 cannot take log of a negative number

03 cannot take SIN or COS of a negative number

04 cannot take SQRT of a negative number

05 only bit positions 0-15 valid

11 invalid device

30 call argument and dummies unequal in number

31 integer formats only

32 number of CALL arguments not as expected

33 invalid argument (out of acceptable range)

40 too many nested repeats in FORMAT

41 OPEN/CLOSE arguments must be integer

42 invalid OPEN mode

43 conflicting file modes

44 attempt to access file not open

45 maximum number of files already open

46 EOFTST or SETEOF with file not open

47 attempt to REWIND file open for output

48 file number already open

49 fatal MDOS related error

50 subscript exceeds allowed range

51 integer overflow

52 real overflow

53 attempt to position LSN past EOF

54 involution value not integer

55 floating point routines missing

56 cannot use BACKSP or DELR after REWIND or WRITE, or on files open for output. Cannot use SETLSN after a WRITE.

63 no driver for READ/WRITE

64 attempt to access device not open

65 maximum number of devices already open

66 MM12 not addressed for I/O

67 improper PPR response position

68 device number already open

69 invalid GPIB device address (must be 1-30)

70 talk address does not match FORTRAN I/O reference number

71 data pool variable not array

72 data pool buffer overflow

73 invalid real-time clock mask

74 attempt to "CALL" a TASK subprogram

75 ACIA framing error

80 argument must be an array

90-99 (Reserved for user)

APPENDIX D

LIBRARY FUNCTIONS

Arguments of functions must be a simple variable, constant, or subscripted variable. Expressions are not allowed. The type of argument (real or integer) must be as shown in the examples (x and y are real; i and j are integer). The function returns a single value result of the type according to function name.

ABS
  Function Type: Real ABS(x)
  Purpose: Returns absolute value of a real number supplied as an argument.

ALOG
  Function Type: Real ALOG(x)
  Purpose: Returns the natural logarithm of "x" (base E), where "x" cannot be negative.

ATAN
  Function Type: Real ATAN(x)
  Purpose: Returns the arctangent (in radians) of the argument.

COS
  Function Type: Real COS(x)
  Purpose: Computes and returns the cosine of "x", where "x" is in radians and not negative.

EXP
  Function Type: Real EXP(x)
  Purpose: Computes and returns e**x.

IABS
  Function Type: Integer IABS(i)
  Purpose: Returns the absolute value of the integer argument.

IAND
  Function Type: Integer IAND(i,j)
  Purpose: Peforms logical AND operation on the arguments and returns the result.

IB
  Function Type: Integer IB(i)
  Purpose: Inputs a single byte found at memory location "i".

  See also function IDB and subroutines BI, BO, DBI, DBO.

IBCLR
  Function Type: Integer IBCLR(i,j)
  Purpose: To clear the "j-th" bit of integer "i" and return the new value of "i". "j" has a range of 0 to 15.

IBSET
  Function Type: Integer IBSET(i,j)
  Purpose: To set the "j-th" bit of integer "i" and return the new value of "i". "j" has a range of 0 to 15.

IBTEST
  Function Type: Integer IBTEST(i,j)
  Purpose: To test the "j-th" bit of integer "i" and return the value of that bit. "j" has a range of 0 to 15.

IDB
  Function Type: Integer IDB(i)
  Purpose: Inputs two bytes found at memory locations "i" and "i+1".

  See also function IB and subroutines BI, BO, DBI, DBO.

IEOR
  Function Type: Integer IEOR(i,j)
  Purpose: Performs the logical exclusive OR operation on the arguments and returns the result.

INOT
  Function Type: Integer INOT(i)
  Purpose: Performs the logical complement of the argument and returns the result.

IOR
  Function Type: Integer IOR(i,j)
  Purpose: Performs the logical inclusive OR on the arguments and returns the result.

IRAND
  Function Type: Integer IRAND(0)
  Purpose: Returns a random integer number.

  See subroutine RNDMZ for further information.

ISHFT
  Function Type: Integer ISHFT(i,j)
  Purpose: Performs the logical shift of integer "i" by "j" bit positions. If "j" is positive, the shift is to the left. If "j" is negative, the shift is to the right. Zeros are shifted in to fill the vacated bit positions. There is no wrap-around of bits; therefore, any absolute value of "j" exceeding 16 guarantees a result of zero.

MADV

Function Type: Integer MADV(i)

Purpose: Returns the memory address of variable "i". The variable argument may be integer, real, or subscripted.

MOD

Function Type: Integer MOD(i,j)

Purpose: Returns the result of "i" modulo "j".

POWER

Function Type: Real POWER(x,y)

Purpose: Computes "x" raised to the "y" power. "x" must not be negative.

SIN

Function Type: Real SIN(x)

Purpose: Computes and returns the sine of angle "x", where "x" is in radians and not negative.

SQRT

Function Type: Real SQRT(x)

Purpose: Computes and returns the square root of "x". "x" cannot be negative.

APPENDIX E

LIBRARY SUBROUTINES

Expressions or functions are not allowed as subroutine arguments. The type (real or integer) and number of arguments must be as shown in the examples (all arguments of subroutines shown are integer).

ATTACH

Subroutine CALL ATTACH(i,j,k,l)

Purpose: To set up an association between an interrupt and a task in a Real-Time system.

Enter: See paragraph 9.7

BACKSP

Subroutine CALL BACKSP(i,j)

Purpose: Backspaces the number of records specified in an MDOS disk file.

Enter: i = FORTRAN I/O file reference number
j = number of records to be backspaced

Exit: i unchanged
j = actual number of records backspaced

BI

Subroutine CALL BI(i,j)

Purpose: To input one byte at address "i" to variable "j".

Enter: i = memory address
j = variable to receive data

Exit: i unchanged
j = one byte of data from memory address "i".

See also: functions IB, IDB and subroutines BO, DBI, DBO.

BO

Subroutine CALL BO(i,j)

Purpose: To output one byte to address "i" from variable or constant "j".

Enter: i = memory address
j = one byte of data to be output (the LS byte of an integer)

Exit: i unchanged
j unchanged

See also: functions IB, IDB and subroutines BI, DBI, DBO.

CNIN

Subroutine CALL CNIN

Purpose: This subroutine inputs from the console keyboard to the I/O buffer defined by the symbol BUF$. When called in this manner, the normal console prompt is printed first.

Exit: Any characters input will be in BUF$ buffer followed by the EOT (hexadecimal 04) character.

Note: May be overridden by KEYIN. See subroutine KEYIN.

CNINNP

Subroutine CALL CNINNP

Purpose: Same as CNIN except no prompt will be issued.

CNOUT

Subroutine CALL CNOUT

Purpose: This subroutine causes the buffer pointed to by the X (index) register to be printed on the console. The first character will be interpreted as the format control character. The output will stop upon encountering the hexadecimal 04 (EOT) character.

Enter: The index register must point to the first character to be output.

Note: Because of the index register requirement, this subroutine will most likely be of value mainly in assembly language programs, although it could be used in conjunction with the MADV function in FORTRAN programs.

CRQS

Subroutine CALL CRQS

Purpose: Clears the request for service for MM12A. See chapter 11, MM12A routines.

DBI

Subroutine CALL DBI(i,j)

Purpose: To input two bytes at address "i" and "i+1" to variable "j".

Enter: i = memory address

Exit: i unchanged
j = two bytes of data from memory address "i" and "i +1".

See also: functions IB, IDB and subroutines BI, BO, DBO.

DBO

Subroutine CALL DBO(i,j)

Purpose: To output two bytes to address "i" and "i+1" from variable or constant "j".

Enter: i = memory address
j = two bytes of data to be output

Exit: i unchanged
j unchanged

See also: functions IB, IDB and subroutines BO, BI, DBI.

DELF

Subroutine CALL DELF(i)

Purpose: Deletes an MDOS diskette file which is presently open.

Enter: i = FORTRAN I/O file reference number

Exit: i unchanged

DELR

Subroutine CALL DELR(i,j)

Purpose: To delete the specified number of records from an MDOS disk file open for input (read), starting at the present position.

Enter: i = FORTRAN I/O file reference number
j = number of records to be deleted

Exit: i unchanged
j = actual number of records deleted

Note: "Deletion" of records means to null-fill them on the diskette.

DEVOFF

Subroutine CALL DEVOFF(i)

Purpose: For I/O devices with drivers which implement this function, to turn off something associated with the particular device. This can only be used for files which have been OPENed with associated drivers.

Enter: i = FORTRAN I/O file reference number

Exit: i unchanged

DEVON

Subroutine CALL DEVON(i)

Purpose: For I/O devices with drivers which implement this function, to turn on something associated with the particular device. This can only be used for files which have been OPENed with associated drivers.

Enter: i = FORTRAN I/O file reference number

Exit: i unchanged

DUMP

Subroutine CALL DUMP(i,j,k,l)

Purpose: Prints a specified area of memory to either the console or line printer for diagnostic purposes.

Enter: i = starting address
j = ending address
k = device number, 101 for console, 102 for line printer
l = identification number which gets printed on the dump

Exit: all parameters unchanged

ENFP

Subroutine CALL ENFP(i)

Purpose: Enables front panel controls of the instrument(s) specified. For use with MM12.

Enter: i = the integer bus address of a single instrument, or an integer array containing one or more bus addresses.

Exit: i unchanged

Note: Refer to chapter 11, MM12 for further information.

ENSRQI

Subroutine CALL ENSRQI

Purpose: Enables IRQ to be generated by MM12 with SRQ. For use with MM12

Note: Refer to chapter 11, MM12 for further information.

```
EOFTST
  Subroutine        CALL EOFTST(i,j)
  Purpose: To detect an end of file (EOF) condition for an MDOS disk file being
           read without aborting in a fatal error.
  Enter:   i = FORTRAN I/O file reference number
  Exit:    i unchanged
           j = 1 for normal condition, or
               2 for end of file indication


ERR
  Subroutine        CALL ERR(i)
  Purpose: Prints an execution time error on the console and stops the program.
  Enter:   i = error number, from 1 to 99.
  Exit:    There is no return from this CALL.


EXIT
  Subroutine        CALL EXIT
  Purpose: To stop execution of a program and return to the operating system.
           This will return to MDOS in a system where the executable program has
           been loaded without the "V" option or executed as a command by MDOS.
           Otherwise, an SWI instruction will be executed for the system to
           assume control at that point. No "STOP" message will be printed in
           either case.


FILTST
  Subroutine        CALL FILTST(i,j)
  Purpose: To test for existence of an MDOS disk file by name.
  Enter:   i = integer array containing the file name
  Exit:    i unchanged
           j = -1 if file does not exist
                0 if drive specified was not ready
               +1 if file was found


FSCALL
  Subroutine        CALL FSCALL(i,j,k,l,m)
  Purpose: To allow calling of MDOS system calls (SCALL) from a FORTRAN program.
           The last argument "m" is optional and may be omitted when not needed.
  Enter:   i = SCALL number
           j = A accumulator value
           k = B accumulator value
           l = X index register value
  Exit:    i unchanged
           j = value of A accumulator upon return from SCALL
           k = value of B accumulator upon return from SCALL
           l = value of X index reg.  upon return from SCALL
           m = (if present) value of C-bit of condition code register.
```

GETCB

Subroutine CALL GETCB(i,j)
Purpose: To find the IOCB address of a designated disk file.
Enter: i = FORTRAN I/O file reference number
Exit: i unchanged
j = IOCB address of this file.

GETLSN

Subroutine CALL GETLSN(i,j)
Purpose: To find the logical sector number currently pointed to in an MDOS disk file.
Enter: i = FORTRAN I/O file reference number
Exit: i unchanged
j = logical sector number

GETRIG

Subroutine CALL GETRIG(i)
Purpose: Group Execute Trigger for MM12. Triggers all devices specified.
Enter: i = the integer bus address of a single instrument, or an integer array containing one or more bus addresses.
Exit: i unchanged

Note: Refer to chapter 11, MM12 for further information.

INITLZ

Subroutine CALL INITLZ
Purpose: Used to initialize I/O devices (IOPKG module) when a MAIN FORTRAN program is not used and the IOPKG module is used. A MAIN program will call this subroutine automatically if needed.

KEYIN

Subroutine CALL KEYIN
Purpose: Provides MDOS .KEYIN SCALL for FORTRAN program input, thus allowing the program to be loaded and run from an MDOS CHAIN control program. This subroutine only needs to be called once in any program to cause it to become effective. If actually executed, no noticeable operation will take place.

Special Considerations: If the user has customized his program, some of the custom features (like a substitute for the "ESCAPE" key) will be overridden. In addition, since the MDOS SCALL of .KEYIN is actually being used, MDOS must be present in the system (not overwritten by a "V" load option), and the ESC key must be followed by a "RETURN" to become effective to stop execution of a program.

LISTEN

Subroutine CALL LISTEN(i)
Purpose: Sends the listen address(es) on the GPIB. For use with MM12.
Enter: i = the integer bus address of a single instrument, or an integer array containing one or more bus addresses.
Exit: i unchanged

Note: Refer to chapter 11, MM12 for further information.

LLO
Subroutine CALL LLO(i)
Purpose: Locks out the front panel controls of the instrument(s) specified. For use with MM12.
Enter: i = the integer bus address of a single instrument, or an integer array containing one or more bus addresses.
Exit: i unchanged

Note: Refer to chapter 11, MM12 for further information.

LPCRLF
Subroutine CALL LPCRLF
Purpose: To perform a CR, LF on the line printer

LPDAT1
Subroutine CALL LPDAT1
Purpose: To print a message string on the line printer without a preceding CR, LF.
Enter: The index register (X) must point to the first character in the string to be output. The string must end with a hex 04.

LPDATA
Subroutine CALL LPDATA
Purpose: To print a message string on the line printer with a preceding CR, LF.
Enter: The index register (X) must point to the first character in the string to be output. The string must end with a hex 04.

LPE
Subroutine CALL LPE(i)
Purpose: Sets up the parallel poll response with the status byte. For use with MM12A.
Enter: i = status byte
Exit: i unchanged

Note: Refer to chapter 11, MM12A for further information.

LPINIT
Subroutine CALL LPINIT
Purpose: Line feeds paper on lineprinter up 6 lines for proper positioning to print first line. The number may be changed by customizing .LPINT to the number desired.

LPOUT
Subroutine CALL LPOUT
Purpose: Prints a buffer on the lineprinter, including interpretation of any control characters.
Enter: The index register (X) must point to the first character in the string to be output. The string must end with a hex 04.

LPQ

Subroutine CALL LPQ(i)

Purpose: Queries the user via the console as to whether a line printer is wanted or not. The value of 101 or 102 is returned depending upon response. This variable can be used in all WRITE statements.

Exit: i = 102 if the response from the console keyboard was either Y or y.
= 101 if any other response was given.

MERED

Subroutine CALL MERED(i,j,k,l,m)

Purpose: Performs an MDOS multisector disk read operation.

Enter: j = starting PSN to be read
k = sector buffer start address
l = number of sectors to be read
m = FORTRAN I/O file reference number

Exit: i = error status (0= no error)
others unchanged

MEWRT

Subroutine CALL MEWRT(i,j,k,l,m)

Purpose: Performs an MDOS multisector disk write operation.

Enter: j = starting PSN to be written
k = sector buffer start address
l = number of sectors to be written
m = FORTRAN I/O file reference number

Exit: i = error status (0= no error)
others unchanged

MLOAD

Subroutine CALL MLOAD(i,j)

Purpose: To load a memory image file and either execute it or return to the FORTRAN calling program.

Enter: i = integer array containing MDOS filename.
j = mode: 0=load disk file into memory and return.
1=load disk file into memory and execute.
2=load and execute with command line saved.
(command line is contained in array "i")

Note: default file sufffix is ".LO", default drive is "0".

MRST

Subroutine CALL MRST

Purpose: Performs a master reset of all devices on the GPIB. For use with MM12.

Note: Refer to chapter 11, MM12 for further information.

PAGE

Subroutine CALL PAGE

Purpose: To issue a form feed character to the lineprinter.

PASCTL
  Subroutine CALL PASCTL(i)
  Purpose: To pass control from controller in charge to another controller in the system. For use by MM12.
  Enter: i = address of other controller
  Exit: i unchanged

  Note: Refer to chapter 11, MM12 for further information.

POLTYP
  Subroutine CALL POLTYP(i)
  Purpose: Sets up polling type with MM12.
  Enter: i = mode: 0=serial poll, 1=parallel poll
  Exit: i unchanged

  Note: Refer to chapter 11, MM12 for further information.

PPR
  Subroutine CALL PPR(i,j)
  Purpose: Parallel poll response. Enters the line position (1-8) and device address in a parallel poll table. For use with MM12.
  Enter: i = device address
          j = line position (1-8)
  Exit: unchanged

  Note: Refer to chapter 11, MM12 for further information.

PRI
  Subroutine CALL PRI(i)
  Purpose: Changes the priority level currently being executed by the Real-Time executive.
  Enter: i = priority level (1-255)
  Exit: i unchanged

QCLEAR
  Subroutine CALL QCLEAR
  Purpose: To clear the Real-Time executive queue of all entries. See paragraph 9.10.1.

QDUMP
  Subroutine CALL QDUMP
  Purpose: Provides a dump of the Real-Time executive queues to either the console or line printer (user is queried for which) for diagnostic purposes.

R05A
  Subroutine CALL R05A(i,j,k)
  Purpose: To perform a read operation on MM05A for both signed and unsigned results.
  Enter: i = FORTRAN I/O file reference number
          j = channel number
  Exit: i unchanged
          j unchanged
          k = result

R05B

Subroutine CALL R05B(i,j,k)

Purpose: To perform a read operation on MM05B for both signed and unsigned results.

Enter: i = FORTRAN I/O file reference number
j = channel number

Exit: i unchanged
j unchanged
k = result

R15AS

Subroutine CALL R15AS(i,j,k,l)

Purpose: To perform a read operation on MM15A for signed results.

Enter: i = FORTRAN I/O file reference number
j = channel number
k = gain (1, 2, 4, or 8)

Exit: i,j,k unchanged
l = result

Note: Refer to chapter 11, MM15A for more details.

R15ASA

Subroutine CALL R15ASA(i,j,k,l)

Purpose: To perform a read operation on MM15A for signed results.

Enter: i = FORTRAN I/O file reference number
j = channel number
k = gain (1, 2, 4, or 8)

Exit: i,j unchanged
k = gain actually used
l = result

Note: Refer to chapter 11, MM15A for more details.

R15AU

Subroutine CALL R15AU(i,j,k,l)

Purpose: To perform a read operation on MM15A for unsigned results.

Enter: i = FORTRAN I/O file reference number
j = channel number
k = gain (1, 2, 4, or 8)

Exit: i,j,k unchanged
l = result

Note: Refer to chapter 11, MM15A for more details.

R15AUA

Subroutine CALL R15AUA(i,j,k,l)

Purpose: To perform a read operation on MM15A for signed results.

Enter: i = FORTRAN I/O file reference number
j = channel number
k = gain (1, 2, 4, or 8)

Exit: i,j unchanged
k = gain actually used
l = result

Note: Refer to chapter 11, MM15A for more details.

R15B

Subroutine CALL R15B(i,j,k)

Purpose: To perform a read operation on MM15B.

Enter: i = FORTRAN I/O file reference number
j = channel number

Exit: i unchanged
j unchanged
k = result

RESETG

Subroutine CALL RESETG(i)

Purpose: Resets the device(s) specified on the GPIB. For use with MM12.

Enter: i = the integer bus address of a single instrument, or an integer array containing one or more bus addresses.

Exit: i unchanged

Note: Refer to chapter 11, MM12 for further information.

RNDMZ

Subroutine CALL RNDMZ(i)

Purpose: To provide a starting seed for the function IRAND. If the argument is 0, the subroutine provides a random seed by adding together all memory bytes from $0000 to $7FFF. If the argument is non-zero, then that value will be used for the starting seed, thus producing the same "random" sequence of numbers upon every program use.

Enter: i = seed (see discussion above)

Caution: Since an argument of 0 causes access (reading) all memory locations in the lower 32K of memory, this could cause problems in systems where certain I/O parts are located in that address range.

Note: The above problem and any uncertainies about the randomness of a starting seed may be overcome by the user providing a random number for the argument (non-zero).

RQS

Subroutine CALL RQS

Purpose: Sends a request for service to the active controller. For use with MM12.

RQS12A

Subroutine CALL RQS12A(i)

Purpose: Sends a request for service to the active controller. For use with MM12A.

Enter: i = status

Exit: i unchanged

RTDDMP

Subroutine CALL RTDDMP

Purpose: Produces a dump of accumulated data from the array in effect (declared by the SETRTD call) on the line printer. Execution of this routine also "clears" the array. (Resets pointers).

Note: Refer to subroutine SETRTD.

RTDOFF
  Subroutine CALL RTDOFF
  Purpose: Ends trace action initiated by RTDON.

  Note: Refer to subroutine SETRTD.

RTDON
  Subroutine CALL RTDON
  Purpose: Starts trace action of Real-Time executive queue dispatches.

  Note: Refer to subroutine SETRTD.

SETEOF
  Subroutine CALL SETEOF(i)
  Purpose: Sets the MDOS disk file pointers to the end of file.
  Enter: i = FORTRAN I/O file reference number
  Exit: i unchanged

SETEOT
  Subroutine CALL SETEOT(i)
  Purpose: Sets the end of transmission (EOT) byte to value specified. This is used with MM12 and MM12A. If two non-zero bytes are specified, these bytes will be sent as termination characters by WRITEG routine. Default is $0D0A (CR,LF).
  Enter: i = termination character(s)
  Exit: i unchanged

SETLSN
  Subroutine CALL SETLSN(i,j)
  Purpose: Sets the LSN (logical sector number) of an MDOS disk file to the value given.
  Enter: i = FORTRAN I/O file reference number
          j = desired LSN
  Exit: unchanged

SETRT
  Subroutine CALL SETRT(i,j,k,l)
  Purpose: To initialize the Real-Time executive system.
  Enter: See paragraph 9.3.1

SETRTD

Subroutine CALL SETRTD(i,j)

Purpose: This subroutine sets up a Real-Time executive queue dispatch logging method. The array to which data is sent and the mode of operation is specified by this routine. This routine may be called more than once in a program, even with different array names and modes. Companion routines are RTDON, RTDOFF, and RTDDMP. The data logged includes the tick value of the real-time clock (TIC$), priority level, stack flag, task address, data bytes passed, lock cell address, and stack address.

Enter: i = integer array name, dimensioned with 1st dimension of 6 and a 2nd dimension large enough to accomodate the number of entries desired.

j = mode: 0=start accumulating data until array is filled, then ignore rest.
1=become circular queue, overwriting first data if necessary.

Exit: unchanged

START

Subroutine CALL START(i,j,k,l)

Purpose: To invoke a task in the Real-Time executive.

Enter: See paragraph 9.5

STARTV

Subroutine CALL STARTV(i,j,k,l,m,n)

Purpose: To invoke a task in the Real-Time executive.

Enter: See paragraph 9.6

TALK

Subroutine CALL TALK(i)

Purpose: Send out the talk address on the GPIB. Used with MM12.

Enter: i = talk address desired

Exit: i unchanged

TRESET

Subroutine CALL TRESET

Purpose: Resets the tick counter in the Real-Time system to zero.

TSETUP

Subroutine CALL TSETUP(i,j)

Purpose: Set up the GPIB for the designated talker and one or more listeners. Used with MM12.

Enter: i = the integer bus address of the talker

Enter: j = the integer bus address of a single instrument, or an integer array containing one or more bus addresses of the listener(s).

Exit: j unchanged

Note: Refer to chapter 11, MM12 for further information.

TSTSRQ
  Subroutine CALL TSTSRQ(i,j)
  Purpose: Tests for service request from the designated device. Used with MM12.
  Enter: i = device bus address
  Exit: i unchanged
         j = code: -1 = parallel poll and device not previously declared with
                        a call to PPR.
                   0 = no SRQ from this device.
                   1 = SRQ from this device.

TTIME
  Subroutine CALL TTIME(i)
  Purpose: Returns the current value of the tick counter (TIC$) in the Real-Time
         executive system. Each count represents 1 time period of the
         Real-Time clock being used. See also TRESET.
  Exit: i = value of tick counter.

UNL
  Subroutine CALL UNL
  Purpose: Causes all devices on GPIB to unlisten. Used with MM12.

UNT
  Subroutine CALL UNT
  Purpose: Causes all devices on GPIB to untalk. Used with MM12.

UNTUNL
  Subroutine CALL UNTUNL
  Purpose: Causes all devices on GPIB to untalk and unlisten. Used with MM12.

W05C
  Subroutine CALL W05C(i,j,k)
  Purpose: Performs the output operation to MM05C.
  Enter: i = FORTRAN I/O file reference number
         j = channel number
         k = value to be output
  Exit: all unchanged

W15C
  Subroutine CALL W15C(i,j,k)
  Purpose: Performs the output operation to MM15C.
  Enter: i = FORTRAN I/O file reference number
         j = channel number
         k = value to be output
  Exit: all unchanged

WAIT
  Subroutine CALL WAIT(i,j,k)
  Purpose: To suspend a currently executing Real-Time task for a period of time,
         allowing other tasks to be executed.
  Enter: See paragraph 9.8

WAITE

Subroutine CALL WAITE(i)

Purpose: To suspend a currently executing Real-Time task until a given event happens, allowing other tasks to be executed.

Enter: See paragraph 9.9

WAITZ

Subroutine CALL WAITZ

Purpose: To suspend a currently executing Real-Time task to allow other tasks of the same or higher priority a chance for execution. This is the same as WAIT with zero time arguments.

WT4CTL

Subroutine CALL WT4CTL

Purpose: Causes the program to loop until the MM12 receives control of the GPIB. For proper operation, this subroutine should always be used before any commands are given on the GPIB by MM12.

APPENDIX F

EXAMPLE FORTRAN PROGRAMS

The following simple program will be used as an example to take the beginning user of MDOS FORTRAN through the steps of compiling and linking.

The sample source program is in a diskette file named RADIUS.SA. This is how the source program appears:

```
*********************************************************
* FOR A GIVEN RADIUS (R), THIS PROGRAM CALCULATES THE
* 2 DIMENSIONAL (CIRCLE) DIAMETER, CIRCUMFERENCE, AND
* AREA.  THE 3 DIMENSIONAL (SPHERE) SURFACE AREA AND
* VOLUME ARE ALSO CALCULATED.
*********************************************************
$1
 PROGRAM DEMO1
100 WRITE(101,9000)
 PI=3.14159
 READ(100,9010) RADIUS
 DIAM=2*RADIUS
 CIRCUM=PI*DIAM
 AREA=PI*RADIUS**2
 SURF=4*AREA
 VOLUM=(4*PI*RADIUS**3)/3
 PRINT 9020,RADIUS,DIAM,CIRCUM,AREA,SURF,VOLUM
 GOTO 100
$1
9000 FORMAT('0ENTER RADIUS')
9010 FORMAT() ; FREE FORMAT READ (IN-LINE COMMENT)
9020 FORMAT(' A RADIUS R= ',F4.2,' GIVES:'/
&    '     DIAMETER     = ',F4.2/
&    '     CIRCUMFERENCE= ',F4.2/
&    '     AREA         = ',F4.2/
&    '     SURFACE      = ',F4.2/
&    '     VOLUME       = ',F4.2)
 END
```

The next step is calling upon the FORTRAN compiler to compile the source program. The MDOS command line and resultant output which appears on the console:

```
=FORT RADIUS;LS

MDOS 6800 RT FORTRAN -  3.10
Copyrighted 1980 by Motorola, Inc.
```

... and since the command line calls for line printer output (L option), the following appears on the line printer:

```
Page 001      MDOS 6800 RT FORTRAN -  3.10

00001 ********************************************************
00002 * FOR A GIVEN RADIUS (R), THIS PROGRAM CALCULATES THE
00003 * 2 DIMENSIONAL (CIRCLE) DIAMETER, CIRCUMFERENCE, AND
00004 * AREA.  THE 3 DIMENSIONAL (SPHERE) SURFACE AREA AND
00005 * VOLUME ARE ALSO CALCULATED.
00006 ********************************************************

00008       PROGRAM DEMO1
00009 100   WRITE(101,9000)
00010       PI=3.14159
00011       READ(100,9010) RADIUS
00012       DIAM=2*RADIUS
00013       CIRCUM=PI*DIAM
00014       AREA=PI*RADIUS**2
00015       SURF=4*AREA
00016       VOLUM=(4*PI*RADIUS**3)/3
00017       PRINT 9020,RADIUS,DIAM,CIRCUM,AREA,SURF,VOLUM
00018       GOTO 100

00020 9000  FORMAT('0ENTER RADIUS')
00021 9010  FORMAT() ; FREE FORMAT READ (IN-LINE COMMENT)
00022 9020  FORMAT(' A RADIUS R= ',F4.2,' GIVES:'/
00023      &   '     DIAMETER     = ',F4.2/
00024      &   '     CIRCUMFERENCE= ',F4.2/
00025      &   '     AREA         = ',F4.2/
00026      &   '     SURFACE      = ',F4.2/
00027      &   '     VOLUME       = ',F4.2)
00028       END

DEFINED Symbols:
Symbol S Addr     Symbol S Addr     Symbol S Addr     Symbol S Addr

ET$R0A X 0800     ET$R01 X 0B00     SURF   D 0078     ET$R0B X 0E00
CIRCUM D 0070     VOLUM  D 007C     ET$R07 X 0900     ET$R08 X 0D00
ET$R09 X 0A00     RADIUS D 0068     PI     D 0064     AREA   D 0074
DIAM   D 006C     ET$R00 X 0C00

Program Size:
CSCT=0000  DSCT=0080  PSCT=01A5  Total=0225
```

Since we had no errors, the next step in the process is linking the object code produced by the above compilation in a file named RADIUS.RO. The following is what appears on the console for this process:

```
=RLOAD
MDOS LINKING LOADER REV  3.01
COPYRIGHT BY MOTOROLA 1977
?BASE
?IF=TEMP
?LOAD=RADIUS
?LIB=FORLB
?OBJA=RADIUS
?MAPF

          NO UNDEFINED SYMBOLS

        MEMORY MAP

          S SIZE  STR  END COMN
          B 0000 0040 0040 0000
          C 0000 2000 2000 0000
          D 01A4 2000 21A3 0022
          P 17AC 21A4 394F 0051

        MODULE NAME BSCT DSCT PSCT
          DEMO1     0040 2000 21A4
          ETRIO     0040 2080 234A
          ET$R0A    0040 20D4 2DB2
          CNIN      0040 20D4 2DBA
          IOBUF     0040 20D6 2E1C
          ET$R00    0040 215C 2E1C
          ET$R01    0040 215C 3106
          RTDUM     0040 215C 362C
          ET$R26    0040 2164 3634
          ERROR$    0040 216E 36B8
          ET$R0B    0040 217A 374A
          LPOUT     0040 217A 3770
          CNOUT     0040 217E 37CE
          IOPKG     0040 2180 37FA
          EXIT      0040 2182 38FC

        COMMON SECTIONS

           NAME  S SIZE  STR
          .SCR$$ D 0008 2182
          .DISKA P 0006 38FF
          .PORTA P 0002 3905
          .INIT  P 0002 3907
          .DELST P 0003 3909
          .FCHRS P 0003 390C
          .KYADR P 0002 390F
          .PRMPT P 0004 3911
          .ADDM  P 0004 3915
          .MPCOM D 0019 218A
          .ERSTK P 0001 3919
```

```
  .LCRLF P 0003 391A
  .LFMFD P 0004 391D
  .LNRDY P 0017 3921
  .XBRKV P 0003 3938
  .XCBRK P 0002 393B
  .CFMFD P 0004 393D
  .CNNUL P 0002 3941
  .CIC   D 0001 21A3
  .IOADR P 000B 3943
  .M12CA P 0002 394E

DEFINED SYMBOLS


MODULE NAME: DEMO1
  MAIN$  P 21A4    STACK$ D 2063

MODULE NAME: ETRIO
  BLFIL$ P 2687    ET$R07 P 234A    ET$R08 P 241B    ET$R09 P 2D2F
  FILE$  D 20C5    HERE9$ P 265F    IN$OUT D 20C6    IO1H$  P 23D7
  P$ADR  D 20D2    X1$    D 20C2

MODULE NAME: ET$R0A
  ET$R0A P 2DB2

MODULE NAME: CNIN
  CNIN   P 2DBD    CNINNP P 2DC8    DBLLF$ P 2E1A

MODULE NAME: IOBUF
  BUF$   D 20D6    BUFSZ$ A 0086    EBUF$  D 215B

MODULE NAME: ET$R00
  ET$R00 P 2E1C    IM$R   A 0000    NEGD$  P 2EEA    SLOG$  P 30D0
  STADT$ P 2F23

MODULE NAME: ET$R01
  ET$R01 P 3106    RM$R   A 0001

MODULE NAME: RTDUM
  ERNUM$ D 2162    IMPRI$ P 362C    PRI$   P 362C    SPND$  P 362C
  WAIT   P 362D    WAITZ  P 3633

MODULE NAME: ET$R26
  ET$R26 P 3634

MODULE NAME: ERROR$
  ERROR$ P 36B8

MODULE NAME: ET$R0B
  CLALL$ P 3766    ET$R0B P 374A

MODULE NAME: LPOUT
  LPCRLF P 37C2    LPDAT1 P 379F    LPDATA P 3788    LPOUT  P 3770
```

```
MODULE NAME: CNOUT
  CNOUT  P 37CE    PDATA$ P 37E7

MODULE NAME: IOPKG
  IN$NE  P 3899    IN$NP  P 3854    IN$NPE P 38F7    INITLZ P 37FA
  LOUTC$ P 38BA    OUTCH$ P 3857    PCRLF$ P 383E    PDAT1$ P 388E

MODULE NAME: EXIT
  EXIT   P 38FC

?EXIT
=
```

At this time, the process is complete and we have an absolute object file named RADIUS.LO on our diskette. We may now execute the program:

```
=LOAD RADIUS;G
```

... and the program prompts us for the radius. To end execution, use the ESCape key on the console.

A few comments are in order for this simple program. We could have used the command of IDON in RLOAD to show the names and identifications of all the modules loaded and encountered in the library. This may be interesting the first few times you go through the process, and can be of some benefit if various errors (such as MDS - multiply defined symbol) occur. Through various RLOAD comands, we could have put the PSCT and DSCT almost anywhere in memory. The BASE command was used to start the loading process at $2000 (above MDOS). We could have named the resultant absolute object file RADIUS.CM, thus making it possible to load and execute as a command.

MASTERMIND

A second example (this one more fun) source program is on the MDOS FORTRAN master product diskette received from Motorola. There are two files - MSTRMIND.SA and GETA1.SA. The first is a FORTRAN source file, while the second is an assembly language source file. The two, when compiled (or assembled) and linked with the FORTRAN library, produce the object code for a game called MASTERMIND. (MASTERMIND is a trademarked name of a game produced by Invicta.)

This program illustrates several features of MDOS FORTRAN. The following is a chain file which may be used to compile, assemble, and link the program. Note that the use of a line printer is assumed - if a line printer is not available, the user must change the "L" in the FORT and RASM command lines to "L=#CN", and the "MO=#LP" to "MO=#CN" in the RLOAD commands.

```
FORT MSTRMIND;ISAL
RASM GETA1;RXL
RLOAD
BASE
IF=TEMP
LOAD=MSTRMIND,GETA1
LIB=FORLB
OBJA=MSTRMIND.CM
MO=#LP
MAPF
EXIT
```

After the user has created the above file with an editor, merely type the MDOS command:

=CHAIN fn

(where fn is the name of the chain file with a CF suffix)

and the process is "automatic". To run the game, type the command "MSTRMIND".

Once the user has produced listings from the above process for reference, the following may be noted:

1. The main program calls upon GETA1 to get one character from the console keyboard and to return a random number. Two methods of passing values are used:

    a. The keyboard character is passed as an argument.
    b. The random number is passed in CSCT or COMMON.

2. The I option was used in the compilation to save nearly 1K bytes of object code memory. Only integer (no real) values were used in the program.

3. The X and Y compile options are illustrated. The user may wish to recompile/re-link with one or both of these options.

4. The BELL control character may be sent to the console, as shown in lines 16 and 61 of MSTRMIND.SA.

5. In GETA1, note that ARG1 on line 37 will contain the address of where the argument value is stored - not the actual value of the argument.

APPENDIX G

LINKING FORTRAN AND ASSEMBLY LANGUAGE PROGRAMS

LINKING FORTRAN MAIN PROGRAMS WITH ASSEMBLY LANGUAGE SUBROUTINES

There are several ways to pass arguments (data) between a FORTRAN program and an assembly language subroutine. The easiest is probably using COMMON in FORTRAN and CSCT in the assembly language program. Keep in mind that integers take 2 bytes, while real numbers take 4 bytes of storage.

Another way is to call an assembly language subroutine from FORTRAN just like any other subroutine. There is no difference in the FORTRAN program (keep in mind the limitation of 13 arguments maximum). The linkage is accomplished in the assembly language subroutine as follows:

1. Use an XDEF followed by the subroutine name. Avoid the '$' and '.' (period) characters in the name because FORTRAN will not allow them in the CALL statement.

2. Use an XREF ET$R16 in the subroutine. Let's assume that we will have a subroutine named SUB23. The first of the assembly language program might look like:

```
NAM SUB23       (this does nOt have to be the same name)
XDEF SUB23
XREF ET$R16
```

3. Set up an area in DSCT of the subroutine for receiving the addresses of the arguments. Suppose there were 4 arguments to be passed. This is how it might look:

```
 DSCT
ARG1 RMB 2 ADDRESS OF 1ST ARGUMENT
ARG2 RMB 2 ADDRESS OF 2ND ARGUMENT
ARG3 RMB 2 ADDRESS OF 3RD ARGUMENT
ARG4 RMB 2 ADDRESS OF 4TH ARGUMENT
```

4. Use the subroutine name as a label at its entry point in PSCT.

5. After the entry point of the subroutine, make a call to ET$R16 to do the work of passing the argument addresses. It should take this form in our example:

```
 PSCT
SUB23 JSR ET$R16
 FDB ARG1
 FDB ARG2
 FDB ARG3
 FDB ARG4
 FCB 4          (this tells the routine there were only 4 arguments)
```

6. You will no doubt make use of the indexed addressing mode to fetch the actual data. Keep in mind that the double bytes in DSCT area contain the ADDRESS of the variable or constant, NOT the actual data.

7. After the FCB 4 in the above example, continue the instructions in the subroutine. The ET$R16 routine will find its way there after it does its work of argument address passing.

8. End the logical conclusion of the subroutine with RTS, and the assembly language program with the END assembler directive.

9. Use the Relocatable Macro Assembler to assemble (RASM with "R" option).

There is a check between the CALL and the subroutine to determine that there is an equal number of arguments being sent and received. The above subroutine would be called like this:

```
CALL SUB23(A1,A2,A3,KK)
```

LINKING FORTRAN MAIN PROGRAMS WITH ASSEMBLY LANGUAGE FUNCTIONS

The main difference between the subroutine and function in MDOS FORTRAN is that a single result is passed back to the calling program by the function.

Upon entry to the function, the index register (X) contains the address of where the result should be placed. Therefore, the usual procedure will be to save the value of the index register first before obtaining the argument addresses via JSR ET$R16. Then, after the necessary calculations are made, the result is stored in memory as addressed by the saved value of the index register. The result will be either 2 or 4 bytes, depending upon the function name as being integer or real.

LINKING ASSEMBLY LANGUAGE PROGRAMS AND FORTRAN SUBROUTINES

Often it is nice to be able to call upon FORTRAN to do certain calculations and I/O to a printer from an assembly language program. Again, arguments may be passed in COMMON/CSCT or with an argument list.

Two precautions: (1) An XREF must be used in the assembly language program to any FORTRAN subroutine name used, and (2) Don't forget to initialize the stack pointer in your assembly language program!!! FORTRAN does it for you in the case of a main FORTRAN program, but the programmer must take the responsibility in cases where he is dealing with only FORTRAN subroutines and functions.

To call a subroutine written and compiled by FORTRAN, use:

```
JSR SUBF      where SUBF is the subroutine name desired
```

To pass arguments, use this combination immediately following the JSR for each argument:

```
FCB xx
FDB yyyy
```

where xx is either $00 or $40, and yyyy is the direct or indirect address of the argument. xx=$00 if yyyy is the actual address of the argument, and xx=$40 if yyyy is an address where the actual address is stored (indirect).

The last argument must have an FCB with bit 1 set. This means a value of either $02 or $42.

If FORTRAN I/O is to be used, the subroutine INITLZ must be called before calling upon any FORTRAN routines using the IOPKG.

LINKING ASSEMBLY LANGUAGE PROGRAMS AND FORTRAN FUNCTIONS

This process is slightly different from linking with FORTRAN subroutines. The only actual difference is that prior to using the JSR to the FORTRAN function, the index register (X) must be loaded with an address of a 2- or 4-byte RAM area where the value of the function will be returned. The 2 or 4 depends upon whether the function is integer or real.

Following is an example of a program using both a function and subroutine:

```
 NAM TEST
 XREF SQRT,PRNT
 DSCT
NUMB FDB $0140,$0000 REAL NUMBER 4.0
ANSWER RMB 4
 RMB 100 STACK AREA
STACK EQU *-1
 PSCT
START LDS #STACK DON'T FORGET THIS!!!
 LDX #ANSWER
 JSR SQRT SQRT IS A FORTRAN FUNCTION
 FCB $02
 FDB NUMB
*ANSWER NOW CONTAINS THE SQRT OF 4.0
 JSR PRNT
 FCB $02
 FDB ANSWER
*ANSWER WAS PRINTED
 SWI
 FCB $1A SCALL .MDENT RE-ENTER MDOS
 END START
```

The accompanying FORTRAN subroutine "PRNT" might look like this:

```
 SUBROUTINE PRNT(VALUE)
 WRITE(101,900)VALUE
900 FORMAT(' THE ANSWER IS ',F5.3//)
 RETURN
 END
```

APPENDIX H

CREATING A LIBRARY OF ROUTINES

A library of various FORTRAN or assembly language routines may be created with the MERGE command. Consider how subprograms call one another before merging. For instance, if routine A calls routine B, A must be merged first. Otherwise, the library must be searched twice. See the Linking Loader manual for more details.

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

```
=MERGE FORLB.RO,PORT.RO,MYLIB.RO
```

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

It is often convenient to create a library of often-used routines - or several libraries. Don't forget to search the library using "LIB=" instead of "LOAD=" during RLOAD. The "LIB=" command only loads modules from the specified file which satisfy unsatisfied XREF names.

APPENDIX I

CHANGING RUNTIME I/O ADDRESSES

For MDOS FORTRAN versions 3.10 and later, a monitor independent I/O package module (IOPKG.RO) is included in the FORTRAN runtime library (FORLIB.RO). The source code for this module (IOPKG.SA) is included on the FORTRAN product diskette. All I/O is referenced to the base addresses of the I/O devices (ACIA, PIA, etc.) as defined in a named common program section (PSCT) labeled ".IOADR".

Use of this module makes the resultant object code not dependent on EXbug and MDOS firmware I/O routines, but rather only the I/O device addresses of the system. Thus the user can easily transport the object code to a micromodule or custom system by changing the I/O device addresses.

Since the monitor independent I/O package is normally used, it should be noted that the echo feature in EXbug 2.X will not function with programs using this I/O module. The output is simply not going through the EXbug subroutines any more.

The named common program sections ".IOADR" and ".CNNUL" are structured as shown here:

```
.IOADR  COMM  PSCT
        FDB   $FCF4     INPUT  ACIA BASE ADDRESS
        FCB   $11       . "     "   CTRL REG BYTE
        FDB   $FCF4     OUTPUT ACIA BASE ADDRESS
        FCB   $11       . "     "   CTRL REG BYTE
        FDB   $EC10     PRINTER PIA BASE ADDRESS
        FCB   $3C       . "      "  CTRL "A" REG BYTE
        FCB   $3C       . "      "   "   "B"  "   "
        FCB   $34       . "      "   "   "A"  "  STROBE

.CNNUL  COMM PSCT
        FCB   0         # NULLS AFTER EACH NON-CR CHAR
        FCB   1         # NULLS AFTER EACH CR     CHAR
```

Notes:

1. Input/output ACIA's are configured as follows:

   BASE+0= status register
   BASE+1= data register

2. Printer PIA is configured as follows:

   BASE+0= "A" side DDR/PDR
   BASE+1= "A" side control register
   BASE+2= "B" side DDR/PDR
   BASE+3= "B" side control register

   "A" side for character output.
   "B" side for status as follows:

   bit 0= 1 if printer ON-LINE
   bit 1= 1 if printer OUT-OF-PAPER
   bits 2-7= don't cares

   CA2 used for data strobe in MDOS version.

3. Null pad values range from zero ($00) through 255 ($FF).

4. The above values are the defaults supplied to correspond with the EXORciser/MDOS environment.

This common section can be changed to match the user's system by any of the following methods:

a. Use the MDOS PATCH command to change the object module after using the linker (RLOAD):

   1. Consult the linker map to obtain the absolute base addresses for .IOADR and .CNNUL common sections.

   2. Use the PATCH command to change the desired locations as required for your system.

   Example: .IOADR= $BC23 and .CNNUL= $BC2E from the linker map. Console ACIA base address in the target system is $ED14, and five nulls are required after CR. No nulls are required after each character. The printer PIA base address is $EC10.

```
=PATCH MYPROG.LO
2400  BD
>BC23/ED,14,,ED,14      change ACIA address
>BC2E,1/5               change CR nulls
>Q                      quit
```

b. Overlay the named common sections .IOADR and/or .CNNUL with the user's values.

   1. Create an assembly language source file which includes the named common sections to be changed. Use "RMB n" to skip over the bytes you do not wish to change.

   2. Assemble the source file using, the proper Macro Assembler (6800/6809) for your system.

   3. Load the resultant module in the linker (RLOAD) just before the OBJA/OBJX command is entered. This causes the user's values to overlay the default system values in the named common sections.

   Example: Same I/O as previous example.

```
              NAM MYIO
              TTL MY I/O DEFINITIONS
              OPT REL
              IDNT 08/14/80 - MY I/O DEFINITIONS
              SPC 3
.IOADR COMM PSCT
       FDB  $ED14  CONSOLE INPUT ACIA
       RMB  1      SAME CTRL VALUE
       FDB  $ED14  CONSOLE OUTPUT ACIA
       SPC  2
.CNNUL COMM PSCT
       RMB  1      SAME NON-CR NULLS
       FCB  5      CR NULL PADDING
       END
```

c. Customize the I/O package source code.

1. Edit a backup copy of the I/O package source code (IOPKG.SA) provided on the disk to match your target system. Instructions to modify the package are included in the source file.

2. Assemble the source file, using the proper Macro Assembler (6800/6809) for your system.

3. Load the customized I/O package module just prior to doing the FORTRAN library search (LIB=FORLB) in the linker (RLOAD).

The monitor independent I/O package module is loaded by default and occupies about 265 bytes of program section (PSCT), including the named common program sections. For certain applications to be installed in read only memory (ROM) where space is tight, this extra memory is not desired. If the necessary I/O subroutines already exist in another ROM (such as MINIbug or MICRObug), these bytes can be saved by any of the following procedures:

a. Define the I/O subroutine addresses in RLOAD.

1. Before entering the FORTRAN library search command (LIB=FORLIB), define the I/O subroutine addresses manually by entering the following (do not enter the parentheses portion):

```
?DEF:IN$NP=$F015        (INCHNP)
?DEF:OUTCH$=$F018       (OUTCH )
?DEF:PCRLF$=$F021       (PCRLF )
?DEF:PDAT1$=$F027       (XPDAT1)
?DEF:LOUTC$=$EBCC       (LIST  )
```

The addresses shown above are for EXbug with MDOS. Using the names given in parentheses, consult the MDOS Equate File Listing and the EXbug Subroutines and Entry Points for specific details. The user can substitute equivalent subroutine addresses that are available in the target system.

b. Create an I/O subroutine address definition module:

1. Create an assembly language source file which defines the symbols shown above in section (a) as global symbols.

2. Assemble the source file using the proper Macro Assembler (6800/6809) for your system.

3. Load the customized I/O package module just prior to doing the FORTRAN library search (LIB=FORLB) in the linker (RLOAD).

Consult Appendix J, "Customizing FORTRAN for Your Target System", to see additional features possible using named common program sections.

Following is information concerning the buffer and use of the I/O routines.

BUF$ identifies the starting location of a 134-byte buffer. Location BUF$ is for carriage control, so input should start at BUF$+1. The I/O drivers must interpret any carriage control character at BUF$. Since the last byte is EOT ($04) and the first byte is for carriage control, a maximum of 132 printable characters is allowed.

Index register is loaded with either the buffer address or the starting address of a string message when entering the console or line printer output routines. User-supplied drivers should not, therefore, reload the index register.

Return to the operating system is used normally after an error is encountered or after the STOP statement is found. The operating system will be MDOS if the .LO program file is produced using the BASE command in RLOAD and is loaded with only the G option at execution time. The module EXIT is called.

The error routine processes the error number information and normally prints it on the console device. The index register will contain the error number in ASCII at the time control is passed to module ERROR.

APPENDIX J

CUSTOMIZING FORTRAN FOR YOUR TARGET SYSTEM

For MDOS FORTRAN versions 3.10 and later, there are several named common program sections (PSCT) that the user can easily overlay to customize the program for a given target system. A brief description of each section follows along with the default assembly listing.

```
***** CHAR EQUATES ******
EOT     EQU     $04
CR      EQU     $0D
CAN     EQU     $18
ESC     EQU     $1B
RUBOUT  EQU     $7F
SPACE   EQU     $20
BELL    EQU     $07
FF      EQU     $0C
LF      EQU     $0A

*** CONSOLE FORM FEED MSG STRING (via PDAT1$)
.CFMFD  COMM    PSCT
FFSTR   FCB     FF,CR,LF,EOT

*** CONSOLE OUTPUT NULL PADDING ***
.CNNUL  COMM    PSCT
        FCB     0       NUMBER OF NULLS AFTER EACH CHAR.
        FCB     1       NUMBER OF NULLS AFTER EACH CR/LF.

* CONSOLE DELETE CHAR STRING (via CNOUT)
*  (can overlay "+,BS,SPACE,BS,EOT" here to erase character on CRT)
.DELST  COMM    PSCT
DELSTR  FCC     "+\"
        FCB     EOT

* CONTROL TEXT FUNCTION CHARACTERS
.FCHRS  COMM    PSCT
DELETE  FCB     RUBOUT
CANCEL  FCB     CAN
ESCAPE  FCB     ESC

* SEE APPENDIX I FOR .IOADR CHANGES
.IOADR  COMM    PSCT
ACIAI$  FDB     .ACIAI   Input ACIA address
CTRLI$  FCB     .CTRLI   Input ACIA ctrl reg byte
ACIAO$  FDB     .ACIAO   Output ACIA address
CTRLO$  FCB     .CTRLO   Output ACIA ctrl reg byte
LPIA$   FDB     .LPIA    Lineprinter PIA address
CTRLA$  FCB     .CTRLA   LP PIA ctrl reg A byte
CTRLB$  FCB     .CTRLB   LP PIA ctrl reg B byte
STRBA$  FCB     .STRBA   LP PIA ctrl reg A strobe
```

```
.ERSTK COMM   PSCT
NUMBER FCB    4         NUMBER OF STACK ENTRIES PRINTED UPON
                        FATAL EXECUTION TIME ERROR


* LP CR.LF message
.LCRLF COMM   PSCT
LCRLF  FCB    CR,LF,EOT


* LPR Form Feed message
.LCRLF COMM   PSCT
LCRLF  FCB    CR,LF,EOT


* LP not ready message (via CNOUT)
.LNRDY COMM   PSCT
NOTRDY FCB    SPACE,BELL
       FCC    "** PRINTER NOT READY"
       FCB    EOT


* LPINIT Subroutine
.LPINT COMM   PSCT
NLINES FCB    6         # of lines to page up


* LINEPRINTER MESG STRING (VIA CNOUT)
.LPQ   COMM   PSCT
MSG1   FCC    " LINEPRINTER"
       FCB    EOT


* CONSOLE PROMPT STRING (via CNOUT)
.PRMPT COMM   PSCT
PROMPT FCC    /+? /
       FCB    EOT
```

If the printer check for break feature is used, it should be noted that multiple PRINTER NOT READY messages may be generated due to the way output is done in several message strings.

```
* LP break feature
* Here when break found (via JMP)
.XBRKV COMM   PSCT
XBRKV  JMP    LWAIT1
*  * User must fix stack pointer
*  * Must use LWAIT1 in case break & user does
*    not overlay - prevents infinite loop.
*
* Here to check for break condition (via JSR)
.XCBRK COMM   PSCT
XCBRK  CLC
       RTS
```

Example:

The following source listing is an example of customizing by overlaying some of the named common PSCT described above.

```
        NAM     PCOMN
        TTL     NAMED COMMON PSCT OVERLAY EXAMPLE
        IDNT    01.00- NAMED COMMON PSCT OVERLAYS
        SPC     2
*** EQUATES ***
SCALL   EQU     $3F
.CKBRK  EQU     $0D
EOT     EQU     $04
BS      EQU     $08
SPACE   EQU     $20
        SPC     3
* DELETE STRING FOR CRT ERASE FUNCTION
*   (SENT VIA FORTRAN CNOUT MODULE)
*   THE FIRST CHAR IS FOR FORMAT CONTROL.
*
.DELST  COMM    PSCT
        FCB     '+,BS,SPACE,BS,EOT
        SPC     3
* FUNCTIONAL CHARACTER DEF'S
*
.FCHRS  COMM    PSCT
        FCB     BS          DELETE CHAR= BACKSPACE
        RMB     1           CANCEL CHAR
        FCB     'Y-$40      ESCAPE CHAR= CTRL+Y
*    PREVENTS ACCIDENTAL TERMINATION FROM
*    HITTING THE "ESC" KEY!
        SPC 3
* PRINTER NOT READY CHECK FOR BREAK FEATURE
*    (SHOWN HERE FOR MDOS ENVIRONMENT)
.XBRKV  COMM    PSCT
        RTS                 HERE WHEN BREAK FOUND
*
.XCBRK  COMM    PSCT
        FCB     SCALL,.CKBRK CHECK FOR BREAK
        RTS                 .     C= 1 IF BREAK
        SPC     1
        END
```

Changing the Size of the I/O Buffer

The I/O buffer contained within the FORLB.RO library is 134 bytes long. This allows an effective length of 132 characters on input. (The first buffer position is normally used for carriage control and the last position is reserved for the EOT control character.)

To change the buffer size, it is necessary to produce a relocatable module, as shown below, and load this module (LOAD=xxxx) before the library search (LIB=FORLB) is done in the linking loader (RLOAD).

```
          NAM    IOBUF
          XDEF   BUF$,EBUF$,BUFSZ$
          DSCT
BUFSZ$    EQU    134 CHANGE THIS VALUE TO ALTER BUFFER SIZE
BUF$      RMB    BUFSZ$
EBUF$     EQU    *-1
          END
```

Changing the Number of "Ports"

The supplied table for PORT I/O allows up to six "ports" to be open at any time. The user may quite easily customize this table for a lesser or greater number. Each entry requires five bytes. The following module may be assembled by RASM as relocatable, and loaded by RLOAD before performing the library search. Change the value "NPORTS" to the desired number.

```
    NRM PTAB$
    XDEF PTABS$,PTABE$
    IDNT SPECIAL PORT I/O TABLE
NPORTS EQU 6    CHANGE THIS NUMBER ONLY
    SPC 1
    DSCT
PTABS$ EQU *
    RMB  5*NPORTS
PTABE$ EQU *
    END
```

Changing the Number/Sectors of Disk Files

The FORLB.RO run-time library supplied with the MDOS FORTRAN compiler allows a maximum of four disk files open at any given time. In addition, the actual read/write access to the disk handles only one sector (128 bytes) of data per access.

The user of MDOS FORTRAN may easily customize the disk I/O to:

1. Allow a maximum of one to nine (or even more) files open at a time.
2. Allow multisector access to the disk.

Trade-offs involve speed of disk I/O versus memory required. Each file requires 41 + n x 128 bytes, where n is the number of sectors. Using multisector disk I/O will often speed up execution of a program considerably.

A source file named DKBUF.SA is contained on the original MDOS FORTRAN diskette. This file contains instructions for changes. Assembling this file requires the use of RASM.CM (Relocatable Macro Assembler). The assembled relocatable module must be loaded before the library search during link time with RLOAD.CM.

APPENDIX K

USING FORTRAN WITH READ-ONLY MEMORY

MDOS FORTRAN has been implemented so that the user may place his program in some form of Read Only Memory (ROM) and operate it in a system other than a development EXORciser.

The requirements for a candidate ROM program to meet are:

1. The program does not use MDOS disk I/O.

2. Use of the "R" option during all FORTRAN module compilations.

3. Any non-FORTRAN modules (such as assembly language) are ROM-able.

The actual division of ROM/RAM comes about during link load time where the user must specify the start addresses of CSCT, DSCT, and PSCT. The ROM-able portion of a FORTRAN program is PSCT. Both CSCT (if used) and DSCT must be assigned to memory containing RAM.

APPENDIX L

SOFTWARE CONSIDERATIONS

M6809 FORTRAN VERSION

The M6800 and M6809 FORTRAN compilers are compatible with the following exceptions to the M6809 version:

U Stack	Initialized by MAIN program unit. Allocated 32 bytes by default (may be changed by OPTION statement). This stack is used by certain execution time routines, particularly in subscript evaluation.
Y Register	This register is used freely in the library routines.
DP Register	Not used or altered. The direct addressing mode is not used by the 6809 library except for MDOS system calls in the case of disk I/O. The old value is saved and restored, so the user may make free use of the DP register.
SWI2,SWI3	These are not used at present.

MEMORY MAP

Any memory not shown on the RLOAD memory load map is not required to be present in the end system, provided disk I/O is not being used at runtime. The full map is obtained through the use of the MAPF command. (Use MO=#LP to obtain map output on the line printer.)

LINK PRECAUTIONS

The real-time FORTRAN library (FORLB.RO) contains several modules with identical symbol definitions (XDEF). Normally, this will cause no problem. However, the assembly language programmer attempting to reference one or more of these symbols may cause the wrong modules from the library to be loaded, resulting in an MDS loader error (multiply defined symbol).

The symbols to be cautious of are:

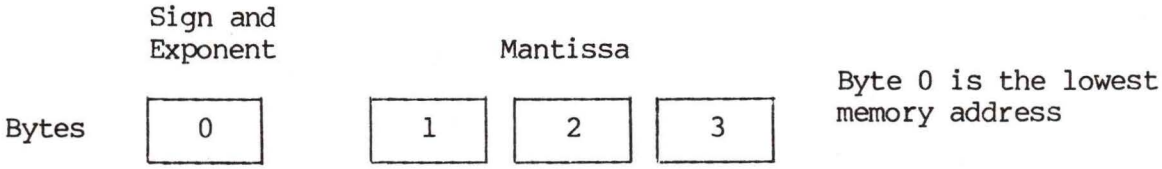
IN$NP, PDAT1$, PCRLF$

If the program is not real-time (i.e., does not call SETRT) and one of the above symbols is referenced in an assembly language subprogram, the user should do the library search (LIB=FORLB) before loading that particular subprogram.

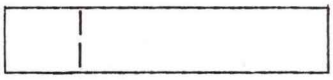
REAL (FLOATING POINT) REPRESENTATION

NOTE

Future releases of MDOS 6800/6809 FORTRAN may change the floating point representation to comply with the IEEE standard. The user is advised to document well any assembly language routines he writes using the present format, as future changes may be required.



Byte 0:



MS bit (7)
is the sign
of the number.
0 for positive
1 for negative

Bits 0-6 - the exponent (base 16) represented in a 7-bit 2's complement form.

Bytes 1-3:

These three bytes represent the mantissa. The hexadecimal point is located to the left of byte 1, and the number is normalized if at least one bit of the upper nibble of byte 1 is set.

EXAMPLES:

Decimal Number	Representation (in hex)
0.0	00 00 00 00
1.0	01 10 00 00
10.0	01 A0 00 00
2.5	01 28 00 00
0.5	00 80 00 00
3215.4	03 C8 F6 66
-1.0	81 10 00 00

INTEGER REPRESENTATION

Integer numbers are represented in 16-bit 2's complement form.

The range of numbers is from -32768 to +32767. The most significant byte is stored at the lower of the two memory addresses.

EXAMPLES:

Decimal Number	Representation (in hex)
0	00 00
1	00 01
3215	0C 8F
-1	FF FE
-32768	80 00
+32767	7F FF

CHARACTER

Literal characters are stored in either 2-byte integer variables or 4-byte real variables. Character data may be placed in variable storage through use of a DATA statement, an assignment, or with a READ statement.

Normally, the characters are left-justified (first character is placed in the lowest memory location) and blank filled (hexadecimal 20) in the event the supplied data is less than the storage area. The exception to this is the R1 format edit code, which right justifies the character with blank fill on the left.

EXAMPLE:

```
DATA I/'AB'/          41 42
DATA A/'ABC'/         41 42 43 20
J='A'                 41 20

DIMENSION FILE(4)
DATA FILE/'TESTDATA.DF:1'/
                      54 45 53 54|44 41 54 41|2E 44 46 3A|31 20 20 20|
```

SUGGESTION/PROBLEM REPORT

Motorola welcomes your comments on its products and publications. Please use this form.

To: Motorola Microsystems
P.O. Box 20912
Attention: Publications Manager
Mail Drop 56Z
Phoenix, Az. 85036

Comments
Product: _____ Manual: _____

Please Print

_____ _____
Name Title

_____ _____
Company Division

_____ _____
Street Mail Drop Phone Number

_____ _____
City State Zip

Hardware/Software Support: (800) 528-1908