MOTOROLA

M68FTN(A1)

MAY 1981

ADDENDUM

TO

M6800/M6809

MDOS FORTRAN

REFERENCE MANUAL

M68FTN(D3)

The following attached pages are replacement pages for the M6800/M6809 MDOS FORTRAN Reference Manual, M68FTN(D3). Revised pages have been marked with change bars to indicate areas of change.

Pages
4-3/4-4
4-5/4-6
5-3/5-4
5-7/5-8
5-15/5-16
7-3
9-1/9-2
10-3/10-4
10-5

Pages
A-1
B-1/B-2
D-1/D-2
E-3/E-4
G-3
I-1/I-2
J-1/J-2
J-3/J-4
J-5
L-1/L-2
L-3/L-4

MICROSYSTEMS

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. For every "IF...."THEN", including "ELSE IF...."THEN", there must be a matching "ENDIF".

EXAMPLE 1:

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

EXAMPLE 2:

```
IF (ITIME1.LT.MAX)  THEN
    <statements>
ELSEIF (ITIME2.LT.MAX)  THEN
    <statements>
ELSEIF (ITIME3.LT.MAX)  THEN
    <statements>
ELSEIF (ITIME4.LT.MAX)  THEN
    <statements>
ENDIF
ENDIF
ENDIF
ENDIF
```

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.

Another usage for DO LOOP's is inside READ, WRITE, and PRINT statements to allow specification of individual elements of an array without implicitly stating them. This form is referred to as IMPLIED DO LOOPS.

EXAMPLE:

```
    DIMENSION PAY(20),RATE(20),IDATE(6)
C  NOTE:  For brevity, the OPEN file #4 statements are deleted.
900 FORMAT( 9(E8.2,2X), 6A1, 5(E8.2) )
    READ(4,900) (PAY(I), I=1,9), IDATE, (RATE(I), I=3,7), M,N
C  The above statement will read 9 values from file #4 into the first 9 elements
C  of the array named "PAY", 6 characters into the array named "IDATE", and 5
C  values into the 3rd through the 7th elements of array named "RATE".
C
    PRINT 910, (PAY(I), I= M,N)
910 FORMAT( 7(F8.2,2X) )
C  The above statement prints the Mth through the Nth element of the array named
C  "PAY" on the console.
```

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.

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. The first character is usually a form control character (see paragraph 5.10.1.3).

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. The first character is usually a form control character (see paragraph 5.10.1.3).

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

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 132 characters unless the I/O buffer size is expanded (see Appendix J).

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).

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 1 to 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 array must be packed ASCII - i.e., it cannot be in an A1 or R1 data format. 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)
```

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.

Examples of this statement are shown in Chapter 10, paragraphs 10.2.1 and 10.2.2, and Chapter 11, paragraph 11.3.1.

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.

NOTE

EXORterm CRT's do not have the IRQ line from ACIA connected. Connecting this line may cause MDOS not to boot due to improper initialization of ACIA by EXbug (XMIT IRQ enabled). Therefore, to boot MDOS, first enter FCF4/11, and then boot MDOS.

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	Interrupt handling
Priorities	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. It is up to the user to control multiple tasks requesting I/O to the same device so as to not interleaf the output.

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.

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 are being used.

```
        NAM     ACIADV
        XDEF    ACIADV
        XREF    BUF$,EBUF$,DV$ADR
        SPC     1
        DSCT
BPTR    RMB     2           BUFFER POINTER STORAGE
        SPC     1
        PSCT
ACIADV  EQU     *
        FDB DEVINT
        FDB DEVTRM
        FDB DEVIN
        FDB DEVOUT
        FDB DEVIP
        FDB DUMMY
        FDB DEVON
        FDB DEVOFF
        SPC     1
* UNIMPLEMENTED VECTORS
DEVTRM  EQU     *
DEVIP   EQU     *
DUMMY   EQU     *
DEVON   EQU     *
```

```
DEVOFF EQU    *
       RTS
       SPC    1
* INITIALIZATION OF ACIA
DEVINT LDX    DV$ADR   GET ACIA ADDRESS
       LDAA   #$03
       STAA   0,X      MASTER RESET
       LDAA   #%00001001
       STAA   0,X      INITIALIZE
       RTS
       SPC    1
* INPUT TO BUFFER FROM ACIA:  TERMINATE ON CR
DEVIN  LDX    #BUF$+1
DEVIN2 STX    BPTR
       LDX    DV$ADR          GET ACIA ADDRESS
DEVIN4 LDAA   0,X             STATUS
       LSRA
       BCC    DEVIN4          READY?
       LDAA   1,X             YES:  GET DATA
       CMPA   #$0D            CR?
       BEQ    DEVIN9          YES:  TERMINATE
       LDX    BPTR            NO:   GET BUFFER POINTER
       STAA   0,X
       INX
       CPX    #EBUF$          END OF BUFFER YET?
       BNE    DEVIN2          .   NE=> NOT YET
DEVIN9 LDAA   #4              EOT
       STAA   0,X             MARK END
       RTS
       SPC    1
* OUTPUT BUFFER TO ACIA:      TERMINATE W/CRLF
*  (IGNORES FORMS CONTROL CHARACTER AT BUF$)
DEVOUT LDX    #BUF$+1
DEVO2  STX    BPTR
       LDAA   0,X             GET CHARACTER
       CMPA   #4              EOT?
       BEQ    DEVO9           YES:  TERMINATE W/CRLF
       BSR    SEND            NO:   SEND CHAR & GET NEXT
       LDX    BPTR
       INX
       BRA    DEVO2
       SPC    1
DEVO9  LDAA   #$0D            CR
       BSR    SEND
       LDAA   #$0A            LF
       BSR    SEND
       CLRA                   NULL
* FALL INTO "SEND" SUBROUTINE!
       SPC    1
SEND   LDX    DV$ADR          GET ACIA ADDRESS
SEND2  LDAB   0,X             STATUS
       LSRB
       LSRB
       BCC    SEND2           READY?
       STAA   1,X             YES:  SEND CHARACTER
       RTS
       SPC    1
       END
```

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. See Chapter 2, paragraph 2.10, for reserved keywords.

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 20 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 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.

The integer value returned can assume the full integer range (-32768 through +32767). A dummy argument (a zero, for example) is necessary to satisfy the requirement that all functions have at least one argument.

See subroutine RNDMZ for further information.

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. The array must be packed ASCII - i.e., it cannot be in an A1 or R1 data format.

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.

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:

```
  NAME TEST
  XREF SQRT,PRNT
  XDEF STACK$
  DSCT
NUMB FDB $0140,$0000 REAL NUMBER 4.0
ANSWER RMB 4
  RMB 100 STACK AREA
STACK$ EQU *-1
************************************************************
* NOTE:  Suggested stack default values (decimal).         *
*        6800:  100 bytes        6809:  140 bytes S-stack *
*                                        32 bytes U-stack *
************************************************************
  PSCT
 START LDS #STACK$ DON'T FORGET THIS!!!
 * MUST INITIALIZE U-STACK IF 6809 BEING USED!
  CLRA SET UP FLAGS ON STACK
  PSHA
  PSHA
  LDX #ANSWER
  JSR SQRT "SQRT" IS A FORTRAN FUNCTION
  FCB $02 ONE ARGUMENT
  FDB ANSWER
 * ANSWER NOW CONTAINS THE SQRT OF 4.0
  JSR PRNT
  FCB $02 ONE ARGUMENT
  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 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
>BC23,0/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
```

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
.LFMFD COMM    PSCT
FFSTR  FCB     CR,LF,FF,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
* Entry:  Stack (6809 S-stack) contains return address.
*         Registers are reserved.
XBRKV  JMP     LWAIT1
*  * User must fix stack pointer
*  * We used 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.) The maximum size is 255 bytes.

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, or RASM09.CM). The assembled relocatable module must be loaded before the library search during link time with RLOAD.CM.

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.)

a. Reducing memory requirements

Examples:

1. Dimensioned variables passed in common to subroutines create a 10-byte table for referencing. Passing a small number of dimensioned variables by argument instead of a large number by common could save some PSCT.

2. Direct initialization of small number of elements of an array could save some PSCT over DO LOOP structure. It is also faster.

```
COMMON ITEMP(3)
ITEMP(1)= 0
ITEMP(2)= 0
ITEMP(3)= 0
```

b. Increasing speed

Examples:

1. IF (A.GT.B.OR.C.LT.D.OR.E.GE.F.OR.G.EQ.0) GOTO 200
   replace with
   IF (A.GT.B) GOTO 200
   IF (C.LT.D) GOTO 200
   IF (E.GE.F) GOTO 200
   IF (G.EQ.0) GOTO 200

2. Assignment of constants and literals to integer variables, including dimensioned ones with coonstant subscripts, is done via a "LDX #","STX <var>".

```
INAME(3)= '.S'  ----->   LDX #$2E53
                         STX $nnnn
```

3. Place repeated, constant calculations outside of DO LOOP's.

```
    DO 20 I= 1,100
10  TREE= 2*3.14 + TRUNK(J) + 10.39
20  RESULT(I)= TREE*5 + I/4
```

Since line 10 does not vary for each iteration of I (J is constant), this line should be moved prior to the loop.

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 MDOS linking loader (RLOAD) error (multiply defined symbol).

The symbols to be cautious of are:

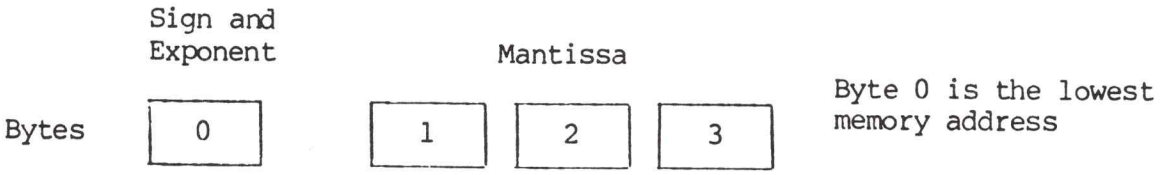
IN$NP, PDAT1$, PCRLF$

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|
```

MOTOROLA *Semiconductor Products Inc.*

P.O. BOX 20912 • PHOENIX, ARIZONA 85036 • A SUBSIDIARY OF MOTOROLA INC.

14379-3 PRINTED IN USA (7/82) MPS 2M