6809 DEVELOPMENT MANUAL

2302-5014-00

PRELIMINARY

February 1982

FutureData
5730 Buckingham Parkway
Culver City, CA 90230

REVISION HISTORY

Title	Number	Date	Notes
6809 Development Manual	2302-5014-00	2/82	Preliminary Software Versions: Assembler 1.0 Linker 5.2 Debugger 2.5

RELATED PUBLICATIONS

UDOS Reference Manual	2301-5002-01
UDOS Programmer's Guide	2301-5006-00
M6809 Cross Macro Assembler Reference Manual	2302-5018-00
MC6809 Preliminary Programming Manual	2302-5029-00

PREFACE

This manual describes the FutureData Assembler, Linker, and Slave Emulator package, which runs on a 2300 Advanced Development System, and which is intended for the development of programs by means of the 6809 processor.

This is a reference manual, not a tutorial; it assumes familiarity with the 2300 ADS, the standard UDOS facilities, and the general concepts of at least one assembly language.

Please note that a **Documentation Reply Card** is inserted at the back of this manual. When you complete and return it, you help us produce better documentation for you.

A **User Registration Card** is included in the set of manuals you receive with your FutureData system. When you complete and return the User Registration Card, you ensure that you will receive all updates and new information for your configuration.

For your convenience, a list of **GenRad/DSD Service Locations** is appended to this manual.

ASSEMBLER

The 6809 Assembler is both a programming language and a language processor which runs under UDOS on FutureData's Advanced Development System. The 6809 Assembler processor accepts as input a source program coded in the 6809 Assembler Language, processes it, and produces a relocatable object module and an assembly listing with diagnostic messages.

This chapter introduces the 6809 Assembler and begins with a summary of its main programming features. Subsequent sections provide general descriptions of the assembly language and the fields which comprise a source program statement. Detailed descriptions illustrate how to invoke the Assembler, and how to specify Assembler options and files.

REFERENCES

Motorola's **MC6809 Preliminary Programming Manual** describes the processor in detail; the assembly language is described in **M6809 Cross Macro Assembler Reference Manual -- M6809XASM(D1) .**

PROGRAMMING FEATURES

The 6809 Assembler provides the programmer with the following features:

- Program sectioning directives which allow flexible control of memory allocation and addressability.
- Assembly control directives which permit repetitive and conditional assembly of a sequence of statements.
- A macro facility which allows a single calling statement to generate a series of in-line instructions and also allows parameter substitution.
- A comprehensive set of expression operators which permits many kinds of arithmetic.

6809 ASSEMBLER LANGUAGE

The 6809 Assembler Language consists of a set of commands and the rules for constructing program statements. There are two classes of commands: mnemonic representations of the Motorola 6809 microprocessor instructions and Assembler directives.

The mnemonic instructions are not described in this manual; the microprocessor manufacturer describes them in the **MC6809 Preliminary Programming Manual** which must be used in conjunction with this manual.

A directive is a command to the Assembler that allows the programmer to assign a program to certain areas in memory, define identifiers, define areas for temporary data storage, place tables or other fixed data in memory, and manage the memory resource. The Assembler's directives are described in this manual.

The Assembler accepts uppercase and lowercase characters as input. The examples in this document are in uppercase for readability.

SOURCE STATEMENTS

A statement is the basic component of an Assembler Language source program. Statements are entered one per line, must not exceed 80 characters, and are composed of a label field, an operation field, an operand field, and a comment field. There are two kinds of statements: instruction statements and directive statements. Instruction statements are of the following form:

```
label:    mnemonic    operand    ; comment
```

A mnemonic is required in an instruction statement. Depending on the specific mnemonic used, the operand may be required, optional, or prohibited. The use of a label or comment field is always optional.

Directive statements are similar in form:

```
name    directive    operand    ; comment
```

A directive is required in a directive statement. Depending on the specified directive used, name and operand may be required, optional, or prohibited. The command syntax descriptions use brackets to indicate optional labels and operands. The use of a comment field is always optional.

The label field consists of either a label or a name. Note that the label in an instruction statement is followed by a colon; the name in a directive statement is not. A label associates a symbolic name with the location of an instruction and can be used as an operand in a JMP or CALL instruction. A label followed by a colon may exist on a blank line. The name in a directive statement performs different functions depending upon the directive being used; the user should not assume that name can be used as an operand in a JMP or CALL instruction.

The operation field consists of either a mnemonic instruction, a macro name, or a directive which identifies the machine operation or Assembler function to be performed. An operation field is required in every statement, except comment lines. An operation field must be separated from the label field by at least one space.

The operand field provides the information needed by the Assembler to perform the designated operation. An operand field consists of one or more identifiers, constants, or expressions separated by commas.

The comment field contains any information the programmer records. A comment field must be separated from the operand field or the operation field by at least one space, and must begin with a semicolon. The comment field is not treated as text and its use does not affect the generation of code.

Fields must be separated by one or more spaces.

If an asterisk is in the first charactrer position on a line, the entire line is treated as a comment. Comment statements are useful for documentation requiring more space than is available in a comment field and for lengthy descriptions such as a program function overview. A line that contains a semicolon as the first non-space character is processed as a comment statement. In a macro library file, comment statements which are not inside macro definitions terminate Assembler processing of the file.

FD ASSEMBLER INPUT

The Assembler accepts Assembler Language source code from a main input file and from a library file. **Figure 1-1** illustrates a sample Assembler input sequence.

```
Label     Operation     Operand          Comment

*
*  Routine to copy a message
*
COPYM     CSECT
          PUBLIC    COPY
COPY:     LDX       6,S              ; Get source text pointer
          LDY       4,S
COPY1:    LDA       ,X+              ; Get a character
          STA       ,Y+              ; Store it
          DEC       2,S              ; Decrement count
          BEQ       COPY1
          LDX       ,S               ; Get return address
          LEAS      8,S              ; Pop parameters, return
          JMP       ,X               ; Return
          END
```

Figure 1-1. Sample Assembler Input

FD ASSEMBLER OUTPUT

Assembler generates an output file consisting of relocatable object code which may be processed by the Linker. The Linker combines the file with other relocatable files and assigns absolute memory addresses.

The Assembler may also print and display a program listing containing each input line, the hexadecimal representation of the object code generated by that line, and other information. A symbol cross-reference listing may be appended to the output file.

Figure 1-2 illustrates a sample output from the Assembler. The first field contains the hexadecimal address within the program segment; the second contains the object output in hexadecimal.

```
Loc.     Assembler  Label  Opcode  Operand          Comments
Counter  Hex Code     Field
                *
                *  Routine to copy a message
                *
 0000           COPYM  CSECT                    ; Start CSECT COPYM
                       PUBLIC  COPY
 0000 AE66      COPY:  LDX     6,S              ; Get source text pointer
 0002 10AE64           LDY     4,S
 0005 A680      COPY1: LDA     ,X+              ; Get a character
 0007 A7A0             STA     ,Y+              ; Store it
 0009 6A62             DEC     2,S              ; Decrement count
 000B 27FF             BEQ     COPY1
 000D AEE4             LDX     ,S               ; Get return address
 000F 3268             LEAS    8,S              ; Pop parameters, return
 0011 6E84             JMP     ,X               ; Return
 0013                  END
```

Figure 1-2. Sample Assembler Output

For each assembly, a symbol table listing is produced showing the memory address of each public (p), each external reference (x), and unreferenced symbol (u), local symbolic label name (no type character):

```
============ 6809 Assembler V5.1 ======================== FutureData =======


   COPY     p0000     COPY1      0005    COPYM   p0000
  No errors
```

INVOKING ASSEMBLER

The Assembler is invoked by executing the following command:

```
JA
```

After a slight pause to facilitate the loading of the Assembler, a list of options appears on the display screen.

ASSEMBLER OPTIONS

The screen in **Figure 1-3** verifies a successful invocation of Assembler and displays a list of options:

```
========= 6809 Assembler V5.1 ====================== FutureData =========

SPECIFY ASSEMBLER OPTIONS:
  (L) - Listing to the screen
  (T) - Truncate lines
  (E) - List errors only
  (S) - Include symbol table
>
```

Figure 1-3. Option Menu

Options may be selected by typing the appropriate letters which are described in **Table 1-1.**

Table 1-1. Assembler Options

Option	Description
L	Displays a program listing on the screen.
T	Truncate lines of the display to 80 characters and limit printer or display listing of the DC directive to one line. If this option is not specified, all lines generated by the DC directive will be output one byte per line.
E	Displays only lines containing errors flagged by the Assembler.
S	Appends the table of symbolic address labels to the end of the relocatable object file.

After zero or more options are selected, enter <RETURN>.

SPECIFYING ASSEMBLER FILE NAMES

Following the entering of options, the Assembler generates a series of prompts. The response to each prompt must meet the requirements listed in **Table 1-2.**

Table 1-2. Assembler File Prompts

Prompt	Requirement
SOURCE FILE:	The file name entered must have an S attribute. This input is required.
MACRO FILE:	This input is optional; enter <RETURN> to bypass.
OUTPUT FILE:	This input is optional; enter <RETURN> to bypass. Any filename entered must have the R attribute (a new file created by the Assembler is automatically be assigned the R attribute).
LISTING FILE:	This input is optional; enter <RETURN> to bypass. Any filename entered must have the S attribute or none at all.

Assembly begins after the programmer responds to the "LISTING FILE:" prompt. If the Assembler recognizes an error, the statement is diagnosed and object code is not generated for that statement.

HALTING THE ASSEMBLY LISTING

While in the program listing phase, the user may halt the assembly listing at any point by pressing the BREAK key. This is useful for viewing the Assembler output as it scrolls on the display. The listing output may be resumed by pressing the BREAK key again.

LANGUAGE ELEMENTS

Input to the Assembler consists of a sequence of characters that are combined to form assembly-language elements. These language elements, which include identifiers, constants, variables, labels, and expressions, comprise program statements which in turn comprise a source program.

IDENTIFIERS

An identifier is a user-defined name which provides a convenient method of identifying constants, variables, labels, and the names of macros, sections, classes, and procedures. Uppercase and lowercase alphabetic characters, numeric characters, and the dollar sign are combined to form identifiers.

An identifier must conform to the following rules:

1. It must begin with an alphabetic character. Although other FD Assemblers permit a dollar sign as the first character of an identifier, this Assembler does not; the dollar sign in the first character position is only used in hexadecimal constants.

2. It may consist of uppercase or lowercase alphabetic characters, numeric characters, and the dollar sign. Other characters are not permitted.

3. It consists of 1 or more characters; characters beyond the eighth are ignored.

4. An identifier may not be an instruction mnemonic, a directive, an expression operator, a register name, or other miscellaneous keyword.

The following identifiers are valid:

```
TEMP3          VALUE          temp27          FOR2$A
```

The following identifiers are invalid:

30DAYS	Identifiers must begin with an alphabetic character.
SECTION 1	Blanks are not allowed.
SUB.2	Periods are not allowed.
$27	Initial dollar sign not allowed.

The asterisk, by itself, is a special, predefined symbol which identifies the value of the current location counter.

CONSTANTS

A constant is a self-defining language element which has no distinguishing characteristics other than its value. Six types of constants are accepted by the Assembler: decimal, hexadecimal, octal, binary, character constant, and string constant. All constants except string constants must be representable in 32 bits, or an "** expression overflow **" error is reported. Constants may have their high-order bits truncated when the value is used in an instruction. This truncation does not cause an error to be reported.

DECIMAL CONSTANTS

A decimal constant is a sequence of numerals ranging from "0" to "9", optionally followed by a "D". The maximum allowed decimal constant is 4294967295D. The following are valid decimal constants:

```
0     16D     1D     1677725
```

HEXADECIMAL CONSTANTS

A hexadecimal constant may consist of the numerals "0" to "9" and the letters "A" to "F". Three formats are provided for writing hexadecimal constants: the Intel "H" suffix, the IBM "X" prefix, and the Motorola "$" prefix

The Intel format requires that a hexadecimal constant begin with one of the numerals "0" to "9" (an extra "0" must be prefixed if the hexadecimal number begins with A through F), and end with the "H" suffix. The following are valid hexadecimal constants expressed in the Intel format:

```
0ACEH     0FFH     18H     0FFFFFFFFFH
```

The IBM format requires that a hexadecimal constant be enclosed in apostrophes, and that an "X" must prefix the first apostrophe. The following are valid hexadecimal constants expressed in the IBM format:

```
X'0'     X'18'     X'ACE'     X'FFFFFFFF'
```

The Motorola format requires that a hexadecimal constant be preceded by a dollar sign. The following are valid hexadecimal constants expressed in the Motorola format:

```
$0       $18       $ACE       $FFFFFFFF
```

OCTAL CONSTANTS

An octal constant may consist of the numerals "0" to "7". Two formats are provided for writing octal constants: the FutureData "Q" suffix and the Motorola "@" prefix

The following are valid octal numbers in the FutureData suffix format:

```
377Q     777341Q
```

The following are valid octal numbers in the Rockwell prefix format:

```
@377     @777341
```

BINARY CONSTANTS

An binary constant may consist of the numerals "0" to "1". Two formats are provided for writing binary constants: the FutureData "B" suffix and the Rockwell "%" prefix. A binary constant must be representable within 32 bits. The maximum allowed binary constant is 11111111111111111111111111111111B. The following are valid binary constantsin the FutureData suffix format:

```
1101B     010B
```

The following are valid binary numbers in the Rockwell prefix format:

```
%1101     %010
```

CHARACTER CONSTANTS

A character constant is a single printable ASCII character preceded by an apostrophe. It has the value of the binary code which represents the character; for example, 'A has the value 41H. A character constant is a numeric constant and can appear with other numeric constants in expressions. The following are valid character constants:

```
'B          ';          '@
```

STRING CONSTANTS

A string constant is a sequence of printable ASCII characters enclosed in apostrophes. A character string constant must not exceed 255 characters in length. The following is a valid string constant:

```
'THIS IS A CHARACTER STRING CONSTANT'
```

Two characters, the apostrophe and the ampersand, must be represented in special ways because of other uses of these characters. A apostrophe must be written as two apostrophes in succession; for example:

```
'TWO''S COMPLEMENT'
```

An ampersand must be written as four ampersands in succession; for example:

```
'OPERATING SYSTEM &&&& AND COMPILER'
```

A null string constant is represented by two contiguous apostrophes, as follows:

```
''
```

String constants must not appear in an expression with numeric constants.

VARIABLES

Assembler recognizes three kinds of data items: constants, variables, and labels. Variables identify data items that are manipulated; they form the operands of arithmetic, logical, and data manipulating instructions. A variable is a named entity possessing value and other attributes; the name is used to denote the associated value. Variables have attributes which specify at what offset within a specific section the variable is defined. A variable is defined with a DB, DW, EXTRN, EQU or DS directive.

LABELS

A label is an identifier that names an instruction, data location, or procedure in the object program; labels form the operands of calls and jumps. Labels are declared using an identifier which may be immediately followed by a colon. A label suffixed with a colon may stand alone on a line or may precede an instruction mnemonic or directive.

EXPRESSIONS

An expression is formed from one or more operands combined with arithmetic, relational, logical, shift, or byte-manipulation operators and may contain parentheses as appropriate. Each individual operand in an expression is called a parameter. A parameter may be a constant, identifier, variable, label, or another expression enclosed in parentheses.

Numbers are represented in 32-bit two's-complement form. A positive number is expressed in binary form and stored right-justified in the byte or word. If the binary form is less than 32 bits long, leading zeros are supplied. If the evaluation of an expression produces a value greater than that representable in 16 bits, the low-order 16 bits are retained and the high-order bits are ignored.

EXPRESSION EVALUATION

Evaluation of an expression produces a single value which must be representable within 32 bits. With two exceptions, every numeric parameter is an arithmetic expression which must be absolute, as opposed to relocatable or external. An absolute parameter is simply a constant; a relocatable parameter is one whose value is a function of the position of the program in its memory space; an external parameter is one whose value is filled in by the Linker.

The exceptions are parameters of the "+" and "-" operators. Using "+" and "-", relocatable and external parameters may be used together in expressions.

The following table gives the valid combinations of parameters and the type of the result:

+ Operator

		Right Operand		
		Absolute	Relocatable	External
L E F T	Absolute	Absolute	Relocatable	External
	Relocatable	Relocatable	Invalid	Invalid
	External	External	Invalid	Invalid

- Operator

		Right Operand		
		Absolute	Relocatable	External
L E F T	Absolute	Absolute	Invalid	Invalid
	Relocatable	Relocatable	Absolute *	Invalid
	External	External	Invalid	Invalid

* Valid only if both relocatables are defined in the same section.

The following program gives examples of valid and invalid uses of the + and - operators:

```
T       CSECT                   ; first test CSECT
A1:     EQU     2
A2:     EQU     4
        EXTRN   E1,E2
        NOP
R1:     NOP
        NOP
R2      NOP
        DW      A1+A2           ; 0 abs + abs
        DW      A1+R2           ; 1 abs + rel
        DW      A1+E2           ; 2 abs + ext
*
        DW      R1+A2           ; 4 rel + abs
        DW      R1+R2           ; 5 rel + rel  **INVALID**

        DW      R1+E2           ; 6 rel + ext  **INVALID**
*
        DW      E1+A2           ; 8 ext + abs
        DW      E1+A2           ; 9 ext + rel  **INVALID**

        DW      E1+E2           ; A ext + rel  **INVALID**


U       CSECT                   ; new CSECT
        NOP
R3:     NOP
        NOP
R4      NOP
        DW      A1-A2           ; 0 abs - abs
        DW      A1-R4           ; 1 abs - rel  **INVALID**

        DW      A1-E2           ; 2 abs - ext  **INVALID**
*
        DW      R3-A2           ; 4 rel - abs
        DW      R4-R3           ; 5 rel - rel  (same CSECT)

        DW      R2-R3           ; 5 rel - rel  **INVALID**  (different
                                                              CSECT)
        DW      R3-E2           ; 6 rel - ext  **INVALID**
*
        DW      E1-A2           ; 8 ext - abs
        DW      E1-R4           ; 9 ext - rel  **INVALID**

        DW      E1-E2           ; A ext - ext  **INVALID**

        END
```

An expression is evaluated according to the precedence levels of the operators, as shown in **Table 1-3**, where 7 indicates the highest level of precedence. The precedence level determines which of two successive operations is performed first.

Expressions are evaluated as follows:

1. Evaluations are performed according to operator precedence. Those with the highest precedence are performed first.

2. Parentheses may be used to control the order of evaluation.

3. Division always yields an integer result; any fractional portion is truncated.

PARENTHESES WITHIN EXPRESSIONS

Multitermed expressions frequently require the use of parentheses to control the order of evaluation. Terms inside parentheses are reduced to a single value before being combined with the other terms in the expression. For example, in the expression

```
ALPHA*(BETA+5)
```

the term BETA+5 is evaluated first, and that result is multiplied by ALPHA.

Expressions may contain parenthesized terms within parenthesized terms. For example:

```
DATA+(HRS/8-TIME*2*(JS+MT))+5
```

In the above example, evaluation begins with the innermost set of parentheses and proceeds to the outermost set. The innermost expression, JS+MT, is evaluated first. Parenthesized expressions may be nested to a maximum of 8 levels.

Table 1-3. Expression Operators And Their Precedence Levels

Type of Expression	Operator	Function	Precedence
Arithmetic	+	Addition	5
	-	Subtraction	5
	*	Multiplication	6
	/	Division	6
	.MOD.	Remainder	6
	-	Negation	5
Relational	=	Equal	4
	<>	Not equal	4
	>	Greater than	4
	>=	Greater than or equal	4
	<	Less than	4
	<=	Less than or equal	4
Logical	.NOT.	Logical negation	3
	.AND.	Logical AND	2
	.OR.	Inclusive logical OR	1
	.XOR.	Exclusive logical OR	1
Shift	.SHL.	Shift left logical	6
	.SHR.	Shift right logical	6
Byte Manipulation	H( )	High-byte isolation	7
	L( )	Low-byte isolation	7
	B( )	Byte swapping	7

ARITHMETIC EXPRESSIONS

Arithmetic expressions are used to express a numeric computation, and evaluation of the expression produces a numeric value. An arithmetic expression consists of one or more operands with arithmetic operators and parentheses.

In an arithmetic expression, two operands may not appear in succession; they must be connected by an operator.

Examples of arithmetic expressions and their evaluations are illustrated in **Figure 1-4.**

A*2	Multiplies the value of A by 2.
A.MOD.B	Returns the remainder after A is divided by B.

Figure 1-4. Examples Of Arithmetic Expressions And Their Evaluations

RELATIONAL EXPRESSIONS

A relational expression is used to compare the values of two operands. Evaluation of a relational expression produces a result with a value of either "0FFFFFFFFH" if the operand satisfies the comparison, or "0" if the comparison is not satisfied.

Relational operators may be used on numbers and character strings. Both parameters of a relational expression must be of the same type.

Character string comparisons proceed character by character along the left and right operand strings. If one character string is shorter than the other, the shorter string is considered less than the longer string.

Relations are not defined on addresses.

Examples of relational expressions and their evaluations are illustrated in **Figure 1-5.**

A<B	Returns "0FFFFFFFFH", if the value of A is less than that of B; otherwise, returns "0".
A-B<>7	Returns "0FFFFFFFFH", if the value A-B is not equal to 7; otherwise, returns "0".

Figure 1-5. Examples Of Relational Expressions And Their Evaluations

LOGICAL EXPRESSIONS

A logical expression is used to express a logical computation. Evaluation of a logical expression produces a result with a value of "TRUE" or "FALSE".

Two logical operators can appear in succession, such as B.AND.(.NOT.A).

Figure 1-6 illustrates the evaluations of logical expressions.

A.OR.B	"TRUE", if either A or B meets the specification; otherwise, "FALSE".
A.AND.B	"TRUE", if both A and B meet the specification; otherwise, "FALSE".

Figure 1-6. Examples Of Logical Expressions And Their Evaluations

SHIFT EXPRESSIONS

A shift expression is used to shift a number a specified number of bits to the left or right.

The general form of the SHL operator is as follows:

```
e.SHL.n
```

The SHL operator shifts the e parameter n bits to the left. The left-most n bits are lost. Zeros are shifted into vacated low-order bit positions. n may be an expression.

The general form of the SHR operator is as follows:

```
e.SHR.n
```

The SHR operator shifts the e parameter n bits to the right. This is a arithmetic right shift. The sign bit of the original e value is shifted into vacated high-order bit positions.

Figure 1-7 illustrates shift expressions and their evaluations.

Assume that A is a one-byte parameter equal to 7 and that B is equal to 2:

A.SHL.B	The .SHL. operator shifts the binary equivalent of 7 two bits to the left; 00000111 becomes 00011100.
A.SHR.B	The .SHR. operator shifts the binary equivalent of 7 two bits to the right; 00000111 becomes 00000001.

Figure 1-7. Shift Expressions And Their Evaluations

BYTE MANIPULATION OPERATORS

Byte manipulation operators include H, L, and B.

H Operator

The H Operator accepts a single, number-valued operand and returns the high-order eight bits of the low-order 16 bits of the value. The general form is as follows:

```
H(number)
```

For example, if ABC is equated to the value X'2E35', the HIGH operator evaluates the following expression to X'2E':

```
H(ABC)
```

L Operator

The L operator accepts a single, number-valued operand and returns its low-order eight bits. The general form is as follows:

```
L(number)
```

For example, if ABC is equated to the value X'2E35', the L operator evaluates the following expression to X'35':

```
L(ABC)
```

B Operator

The B operator accepts a single, number-valued operand and returns a 16-bit value with the high- and low-order 8-bits reversed. The high order 16 bits of the 32-bit value are set to zero. The general form is as follows:

```
B(number)
```

For example, if ABC is equated to the value X'2E35', the B operator evaluates the following expression to X'352E':

```
B(ABC)
```

EFFECTIVE ADDRESS SYNTAX

The allowed forms of memory-address operands which correspond to 6809 addressing modes are:

1) Inherent

Several instructions have no addressing options at all, as in

```
MUL
```

2) Accumulator

Accumulator addressing refers to a 6809 accumulator, whose name is written as an opcode suffix, as in

```
CLRA
```

3) Immediate

Immediate addressing refers to the location(s) following the instruction opcode. The immediate value is written as an operand preceded by a #, as in

```
ADDD   #1234
```

4) Absolute Addressing

Absolute addressing refers to a 16-bit address in the 6809's address space. There are three forms of such an address:

a) Direct

Direct addressing is a shorthand; it uses the direct page register as the high-order byte of the address, so that only the low-order byte need be included in the instruction. A direct address is written as an operand preceded by a <, as in

```
LDD    <LOC
```

b) Extended

Extended addressing uses a 16-bit address which is included in the instruction, contained in the bytes following the opcode. An extended reference is written as an operand preceded by a >, as in

```
LDB    >LOC2
```

c) Extended Indirect

Extended indirect addressing uses a 16-bit address which is included in the instruction, contained in the bytes following the opcode, as the address of the effective address. An extended indirect reference is written as an operand enclosed in square brackets, as in

```
ADDD    [LOCA]
```

5) Register

Register addressing refers to 6809 registers as the instruction's operands, as in

```
TFR    DP,A
```

6) Indexed

There are five indexable registers on the 6809: X, Y, U, S, and PC. The indexed references generate an effective address in various ways, combining the contents of an indexable register with instruction bytes or other register contents.

The indexing options are:

- constant-offset
- accumulator offset
- auto-increment/decrement
- indirection

a) Constant-Offset Indexed

Constant-offset indexing uses an optional two's-complement instruction offset with an indexable register to form the effective address. Such an address is written as

```
LDA    ,X          ;offset = 0
LDB    17,U        ;offset = 17
LDA    LABEL,PC
```

b) Constant-Offset Indexed Indirect

Constant-offset indexed indirect addressing uses an optional two's-complement instruction offset with an indexable register to form the address of the effective address. Such an address is written as

```
ADDD    [,U]       ;offset = 0
LDD     [SAM,PC]   ;offset = SAM
```

c) Accumulator Indexed

Accumulator indexed addressing uses an accumulator with an indexable register to form the effective address. The accumulator content is treated as a two's complement value. Such an address is written as

```
LDA    D,Y
```

d) Accumulator Indexed Indirect

Accumulator indexed indirect addressing uses an accumulator with an indexable register to form the address of the effective address. The accumulator content is treated as a two's complement value. Such an address is written as

```
LDA    [D,Y]
```

e) Auto-Increment

Auto-increment addressing uses the content of the indexable register as the effective address. After the memory access, the register's content is incremented by one or two. Such an address is written as

```
LDA     ,Y++        ;increment by 2
LDB     ,X+         ;increment by 1
```

f) Auto-Increment Indirect

Auto-increment indirect addressing uses the content of the indexable register as the address of the effective address. After the memory access, the register's content is incremented by one or two. Such an address is written as

```
LDA     [,Y++]      ;increment by 2
LDB     [,X+]       ;increment by 1
```

g) Auto-Decrement

Auto-decrement addressing uses the content of the indexable register as the effective address. Before the memory access, the register's content is decremented by one or two. Such an address is written as

```
LDA     ,--Y        ;decrement by 2
LDB     ,-X         ;decrement by 1
```

f) Auto-Decrement Indirect

Auto-decrement indirect addressing uses the content of the indexable register as the address of the effective address. Before the memory access, the register's content is incremented by one or two. Such an address is written as

```
LDA     [,--Y]      ;decrement by 2
LDB     [,-X]       ;decrement by 1
```

7) Relative

Short relative addressing adds to the PC the value of an instruction byte to produce the effective address. The instruction byte is treated as a two's-complement value. Such an address is written as

```
BRA    TARGET
```

The assembler will automatically select short relative addressing for backward references which are within 129 bytes; to force short relative addressing for a forward reference, the address is written with a < as

```
BRA    <LABEL
```

8) Long Relative

Long relative addressing adds to the PC the value of an instruction word to produce the effective address. The instruction word is treated as a two's-complement value. Such an address is written as

```
LBRA   TARGET
```

TEMPORARY RESTRICTIONS ON DO, IF, DEFL, DEFG

The Assembler evaluates all expressions using 32-bit arithmetic and logical operations. The current version of the Assembler, however, restricts the way that expression operands of the DO, IF, DEFL, and DEFG directives are used.

Number values assigned to set symbols by DEFL and DEFG are stored by the Assembler as the least-significant 16 bits of the value of the evaluated expression. This can cause unexpected results unless special consideration is given. For example, in the following sequence of statements:

```
ASYM    DEFL    65537
        JMP     L&ASYM
```

the second statement will assemble as:

```
        JMP     L00001
```

rather than the expected result of:

```
        JMP     L65537
```

Similarly, the actions of the IF and DO directives are determined by only the least-significant 16 bits of their operand value. The following IF statement:

```
        IF      1.EQ.65537
```

will be taken as true (the statements following the IF will be assembled rather than skipped). The following DO sequence will assemble only 2 times:

```
        DO      65538
        .
        .
        .
        ENDDO
```

VARIABLE AND IDENTIFIER DEFINITION

A variable can represent an 8-, 16-, or 32-bit data item. An identifier can represent an assigned value or the current assembly location. In addition, an identifier may be made available to other modules. **Table 1-4** lists the variable and identifier defining directives and their functions.

Table 1-4. Variable And Identifier Defining Directives

Directive	Function
DB	Defines a variable as type BYTE and initializes one or more storage units.
DW	Defines a variable as type WORD and initializes one or more storage units.
EQU	Permanently assigns a value to an identifier.
PUBLIC	Specifies which identifiers defined in the current module are made available to other modules.
EXTRN	Specifies identifiers which are defined in other modules but referenced in the current module.
NAME	Assigns a name to the object module generated by the assembly.

DEFINING BYTE VARIABLES

The DB directive defines a variable as type BYTE and initializes one or more storage units.

LABEL	OPERATION	OPERAND
[name][:]	DB	expression[,...]

name — In an absolute or relocatable section, name is defined as the address of a variable of type BYTE.

expression — Specifies any valid expression.

EXAMPLE

```
HUNDRED  DB    100                 Defines a byte with the value 100.

MSG:     DB    'MESSAGE'           Defines a character string named MSG.

MSG      DB    'MESSAGE',13,10     Defines an ASCII character string named
                                   MSG followed by ASCII CR and LF.
```

USAGE NOTES

The DB directive allows the programmer to define 8-bit memory locations that are initialized to specified values. This data may include messages, names, lookup tables, minimum or maximum values, masking patterns, or any data required by the program. Addresses are not valid operands of a DB directive.

A DB directive such as:

```
TEN     DB     10
```

places the 8-bit value 10 in the next available memory location and assigns that location the name TEN.

The DB directive may be used to define character strings that are stored byte by byte as written, that is, left to right.

This directive may also be spelled "FCB" for compatibility with the Motorola directive which performs the same function.

Multiple operands of a DB directive are stored in successive memory locations. Data are stored according to the following rules:

1) ASCII character strings are stored one byte per character. A sequence written as two consecutive apostrophes anywhere within the string's outer apostrophes is stored as a single apostrophe. A sequence written as four ampersands is stored as one ampersand.

2) Defined or external number values are stored truncated to the least significant byte only.

3) The use of relocatable address operands to DB is not recommended unless the H(expression) or L(expression) constructs are used.

4) The H(expression) or L(expression) constructs may be used to select the most-significant or least-significant byte of 16-bit defined or external numbers.

5) The use of a colon after a label is a matter of style and is optional.

DEFINING WORD VARIABLES

The DW directive defines a variable as type WORD and initializes one or more storage units.

LABEL	OPERATION	OPERAND
[name][:]	DW	expression[,...]

name	In an absolute or relocatable section, name is defined as the address of a variable of type WORD.
expression	Specifies any valid expression.

EXAMPLE

FEF:	DW	OFEH	Defines a word named FEF with a hexadecimal value representing 254 decimal.
NUMBER	DW	1234H	Defines a word named NUMBER and stores 34H in NUMBER and 12H in NUMBER+1
REV	DW	B(1234H)	Defines a word named REV and stores 12H in REV and 34H in REV+1.
CHARS	DW	'AB'	Defines a word named CHARS and stores the ASCII characters 'AB' in CHARS as 4241H.
OFFAB	DW	AB	Defines a word named OFFAB and stores the segment-relative (relocatable) offset of variable AB in OFFAB.

USAGE NOTES

The DW directive allows the programmer to define 16-bit storage locations that may be optionally initialized to specified values. This data may include numbers, address offsets, or any data required by the program. If an address is used as the operand of a DW directive, that address' relocatable offset from the beginning of its containing section is stored.

This directive may also be spelled "RMB" for compatibility with the Motorola directive which performs the same function.

ASSIGNING A PERMANENT VALUE TO AN IDENTIFIER

The EQU directive assigns a permanent value to an identifier.

LABEL	OPERATION	OPERAND
name[:]	EQU	expression

name	Specifies any valid identifier.
expression	Specifies any expression that is valid as a single operand of either a machine instruction or a data definition directive, including forward references, external symbols, expressions, addresses, address expressions, and other EQUs.

EXAMPLE

```
HERE    EQU   *      Assigns the value of the current location counter to
                     the identifier HERE.

LAST:   EQU   1000   Assigns the value 1000 to the identifier LAST.

START   EQU   ST+1   Assigns the value of ST+1 to the identifier START.
```

USAGE NOTES

The EQU directive requires a label field since the function of the directive is to define the meaning of the name in the label field. The symbol "name" is assigned the value of expression by the Assembler. Whenever the symbol "name" is encountered subsequently in the assembly, this value is used.

The name defined in an EQU directive may not be redefined.

MAKING SYMBOLS AVAILABLE TO OTHER MODULES

The PUBLIC directive specifies which identifiers or labels defined in the current module are made available to other modules.

LABEL	OPERATION	OPERAND
	PUBLIC	symbol[,...]

symbol — Specifies a variable or label defined anywhere in the current module. The following are not allowed in a PUBLIC directive: the names of macros, macro parameters, local symbols, DEFG and DEFL identifiers, and identifiers equated to expressions that contain externally defined terms or character strings longer than two characters.

EXAMPLE

```
PUBLIC    CNT
```

Specifies that the variable CNT be made available to other modules.

USAGE NOTES

Any identifier or label declared PUBLIC which is not defined in the current module produces a diagnostic message.

MAKING EXTERNAL SYMBOLS AVAILABLE TO THE CURRENT MODULE

The EXTRN directive specifies which identifiers or labels defined in other modules are used by the current module.

LABEL	OPERATION	OPERAND
	EXTRN	symbol[,...]

symbol — Specifies an identifier or label defined in a module other than the current module.

EXAMPLE

```
EXTRN   CNT,ALOOP
```

Identifies a variable CNT which is defined in another module; CNT is made available for use in the current module. Also identifies a label named ALOOP which is defined in another module.

USAGE NOTES

The following example illustrates the use of the EXTRN directive:

```
SAM     CSECT
        EXTRN  LNGTH
         .
         .
         .
              LDA       LNGTH      ; get string's length
        MUL
         .
         .
         .
        END
```

NAMING THE OBJECT MODULE

The NAME directive assigns a name to the object module generated by the assembly.

LABEL	OPERATION	OPERAND
	NAME	module

module	Specifies a valid identifier that is not greater than 8 characters.

EXAMPLE

NAME	MOD13	Assigns the name MOD13 to the program module that follows.

USAGE NOTES

If used with the FD Linker, the NAME directive allows the programmer to combine several different assembly modules into a single load module for execution. The module name assigned with the NAME directive may be used to control storage allocation and linking. This name is stored in the object module and is not necessarily the same as the name of the file that contains the module. The module name specified appears in the link map produced by the FD Linker as the defining module name for sections and PUBLIC symbols and is used in the SECU debugger environment to distinguish among similarly named local symbols.

For compatibility with Motorola, the NAME directive may be spelled NAM.

SECTIONING

The 6809 Assembler provides sectioning facilities to control the placement of instructions and data in memory, thereby aiding in program structuring and memory management.

Sectioning allows the definition of various named areas of instructions, constant data, and variable data. A section is the smallest relocatable unit of memory; every instruction and data item must lie within a section. Each section defines a separate location counter. At least one section must be declared per assembly module; there is no arbitrarily imposed limit, other than overall symbol table capacity, for the number of sections defined in an assembly module.

Sections may be defined for read-only data, working-storage data, the stack, the main program, shared (reentrant) subroutines, interrupt vectors, interrupt routines, or other purposes. Code and data may be mixed in a section, although this practice is not always advisable.

A section can be absolute or relocatable.

An absolute section is assembled, linked, and loaded at an absolute address specified at assembly time. Absolute sections are useful for defining instructions to be executed in the event of an interrupt, or other hardware-dependent code or data.

A relocatable section is assembled such that its location in memory is determined at a later time by the Linker. Most program sections are relocatable.

Table 1-5 lists the sectioning directives and their functions.

Table 1-5. Sectioning Directives

Directive	Function
ASECT	Specifies the beginning of an absolute section.
CSECT	Specifies the beginning of a relocatable section.

DEFINING AN ABSOLUTE SECTION

The ASECT directive specifies the beginning of an absolute section or specifies assembly to an existing absolute section.

LABEL	OPERATION	OPERAND
	ASECT	

EXAMPLE

```
ASECT           Defines a new absolute section starting at 80H.
ORG   80H
```

USAGE NOTES

An absolute section begins with an ASECT directive and terminates with the next ASECT, CSECT, or END directive.

By coding another ASECT directive, it is possible to assemble into a previously defined absolute section after intervening relocatable sections. When switching assembly to a previously-defined absolute section, assembly resumes at the end of that absolute section.

Statements that generate object code must be preceded by an ASECT or CSECT directive; there is no default section.

DEFINING A RELOCATABLE SECTION

The CSECT directive specifies the beginning of a relocatable section or specifies assembly to an existing relocatable section.

LABEL	OPERATION	OPERAND
name	CSECT	

name — Specifies a valid label, which is not greater than 15 characters, by which the section is to be known to the Assembler and the Linker.

EXAMPLE

```
SEC5    CSECT               Defines a new relocatable section named SEC5.
          .
          .
          .
SEC6    CSECT               Defines a new relocatable section named SEC6.
          .
          .
          .
SEC5    CSECT               Switches assembly to relocatable section SEC5,
          .                 which was previously defined.
          .
          .
SEC6    CSECT               Switches assembly to relocatable section SEC6,
          .                 which was previously defined.
          .
          .
          .
```

USAGE NOTES

A relocatable section begins with a CSECT directive and terminates with the next ASECT, CSECT, or END directive.

Assembly of a newly defined relocatable section begins at offset zero.

By coding another CSECT directive containing the name of an existing relocatable section, it is possible to assemble into a previously defined relocatable section after intervening sections. When switching assembly to a previously defined relocatable section, assembly resumes at the end of that relocatable section.

Statements that generate object code must be preceded by an ASECT or CSECT directive; there is no default section.

LOCATION COUNTER

The location counter specifies the storage location to be assigned next. The current location-counter value is available in an expression by using the special symbol *. **Table 1-6** lists the location counter directives and their functions.

Table 1-6. Location Counter Directives

Directive	Function
ORG	Sets the location counter for the current section to a specified value.
DS	Adds a specified value to the current location counter to reserve storage.

When a section is defined, the Assembler initializes that section's location counter to zero.

SETTING THE LOCATION COUNTER

The ORG directive sets or resets the location counter for the section in which it occurs and thereby specifies the location at which subsequent statements are assembled.

LABEL	OPERATION	OPERAND
	ORG	expression

expression — In an ASECT, specifies an absolute number, an expression that can be evaluated to an absolute number, or an address within the ASECT.

In a CSECT, specifies an absolute number (offset within the current section), an expression that can be evaluated to an absolute number, or an address within the current section.

Forward and external references are not allowed.

EXAMPLE

```
ORG   1000      Sets the location counter to offset 1000.

ORG   ADDRS     Sets the location counter to the value of ADDRS.
```

USAGE NOTES

If an ORG directive is not the first statement of a section, assembly begins at location zero, relative to the address at which the section is linked.

The expression operand of an ORG directive must not contain forward or external references. If the expression evaluates to an address, that address must be defined within the current section.

RESERVING STORAGE

The DS directive reserves and optionally names a memory area consisting of a specified number of bytes.

LABEL	OPERATION	OPERAND
[name]	DS	expression

name — Specifies a valid identifier which names the memory area.

expression — Specifies an absolute number or an expression that can be evaluated to an absolute number which represents the number of contiguous bytes to be reserved.

EXAMPLE

```
WORDS   DS   20       Defines a memory area 20 bytes long and names it
                      WORDS.

SPACE   DS   TEN+5    Defines a memory area TEN+5 bytes long and names it
                      SPACE.
```

USAGE NOTES

The value of expression specifies the number of contiguous memory bytes to be reserved for data storage by advancing the location counter for the section in which the DS directive occurs. The DS directive does not assemble any data values into the reserved bytes.

If an identifier is used as the expression or a term of the expression, it must be defined before the DS directive is encountered and may not be an external or address-valued symbol.

For compatibility with Motorola, the DS directive may be spelled RMB.

ASSEMBLY CONTROL

Program assembly proceeds sequentially from one statement to the next unless assembly control directives are used to alter the sequential order of assembly or to terminate program assembly. **Table 1-7** lists the program control directives and their functions.

Table 1-7. Assembly Control Directives

Directive	Function
IF	Identifies the beginning of a block of code which is included or excluded at assembly time depending on the logical value of an expression.
ENDIF	Identifies the end of an IF block.
ELSEIF	Used in conjunction with an IF directive to test an alternate condition within the current nesting level.
ELSE	Used in conjunction with an IF directive to indicate the last alternative within the current nesting level.
EXITIF	Causes all statements preceding the closing ENDIF directive in the current or specified IF block to be ignored.
DO	Identifies the beginning of a block of code which is repetitively assembled zero or more times.
NEXTDO	Causes the Assembler to perform immediately the next iteration of the current or named DO block.
EXITDO	Terminates processing of the statements within the current or named DO block.
END	Indicates the end of a source program.

CONDITIONAL ASSEMBLY

The IF directive identifies the beginning of a block of code which is included or excluded at assembly time depending on the logical value of a specified expression.

LABEL	OPERATION	OPERAND
[name]	IF	expression

name — Specifies a valid identifier which may only be used to name an IF block for reference by an EXITIF directive.

expression — Specifies a number-valued expression. Forward, external, and address references are not permitted.

EXAMPLE

```
DECIDE   IF   A=B
```

Identifies the beginning of a sequence of statements which is included in the assembly if A equals B.

USAGE NOTES

Programming problems often require the user to specify two or more courses of action and a means of deciding which course to assemble. The IF and ENDIF directives identify the beginning and end of a sequence of statements which is or is not assembled depending on the value of a specified expression. The following illustrates the simplest form of an IF block:

```
[IF block name]     IF    expression
                    .
                    .     Statements to assemble if the expression is true.
                    .
                    ENDIF
```

If the expression contained in the operand field evaluates to a logical true (a non-zero value), the statements between the IF and ENDIF directives are assembled. If the expression evaluates to a logical false (a zero value), the statements contained in the IF block are not assembled.

An ENDIF directive must end each IF block. For compatibility with Motorola, the ENDIF directive may be spelled ENDC.

IF blocks may be nested within other IF blocks, as follows:

```
[IF block name-1]     IF     expression-1
                      .
                      .
                      .
[IF block name-2]     IF     expression-2
                      .
                      .
                      .
                      ENDIF
                      .
                      .
                      .
                      ENDIF
```

IF blocks may be nested to any level.

The name assigned to an IF block may only be used in an EXITIF directive; it may not be used in a call, a jump, or any other context.

TESTING AN ALTERNATE CONDITION

The ELSEIF directive is used in conjunction with an IF directive to test an alternate condition within the current nesting level.

LABEL	OPERATION	OPERAND
	ELSEIF	expression

expression — Specifies a number-valued expression. Forward, external, and address references are not permitted.

EXAMPLE

```
ELSEIF   A=B        Defines the beginning of a sequence of statements
                    which is assembled if the expression in the preceding
                    IF directive is false and A equals B.
```

USAGE NOTES

One or more ELSEIF directives may be used within an IF block. The following illustrates the placement of ELSEIF directives within an IF block:

```
[IF block name]     IF          expression-1
                    .           Statements to assemble if expression-1 is
                    .           true.
                    .
                    ELSEIF      expression-2
                    .           Statements to assemble if expression-1 is
                    .           false and expression-2 is true.
                    .
                    ELSEIF      expression-3
                    .           Statements to assemble if expression-1 and
                    .           expression-2 are false and expression-3 is
                    .           true.
                    ENDIF
```

If the expressions of the IF and preceding ELSEIF directives are false and the current ELSEIF expression is true, the statements following the true expression up to the next ELSEIF, ELSE, or ENDIF directive are assembled. If the expressions of the IF and the preceding ELSEIF directives are false and the current ELSEIF expression is also false, the statements following the false expression up to the next ELSEIF, ELSE, or ENDIF directive are not assembled.

TESTING THE LAST CONDITION

The ELSE directive is used in conjunction with an IF directive to indicate the last alternative within the current nesting level.

LABEL	OPERATION	OPERAND
	ELSE	

EXAMPLE

ELSE	Defines the beginning of a sequence of statements which is assembled if the expression in the IF and preceding ELSEIF directives are false.

USAGE NOTES

The following illustrates the placement of an ELSE directive within an IF block:

```
[IF block name]     IF          expression-1
                    .           Statements to assemble if expression-1 is
                    .           true.
                    .
                    ELSEIF      expression-2
                    .           Statements to assemble if expression-1 is
                    .           false and expression-2 is true.
                    .
                    ELSE
                    .           Statements to assemble if all preceding
                    .           expressions are false.
                    .
                    ENDIF
```

If the expressions in the IF and preceding ELSEIF directives are false, the statements following the ELSE directive up to the ENDIF directive are assembled. If any of the expressions that precede the ELSE directive are true, the statements following the ELSE directive up to the ENDIF directive are not assembled.

EXITING AN IF BLOCK

The EXITIF directive causes all statements following it and preceding the closing ENDIF directive in the current or specified IF block to be ignored.

LABEL	OPERATION	OPERAND
	EXITIF	[name]

name — Specifies the name of an outer IF block to be exited. If not specified, the current IF block is exited.

EXAMPLE

```
EXITIF   DECIDE        Exits an IF block named DECIDE.
```

USAGE NOTES

One or more EXITIF directives may be used within an IF block. The following illustrates the placement of an EXITIF directive within an IF block:

```
name-1      IF        expression-1
            .
            .         Statements to assemble if expression-1 is true.
            .
[name-2]    IF        expression-2
            .
            .         Statements to assemble if expression-2 is true.
            .
            EXITIF    name-1
            ENDIF
            .         Statements to assemble if expression-2 is false
            .         and expression-1 is true.
            .
            ENDIF
```

If expression-1 and expression-2 are true, the EXITIF directive, which terminates the name-1 IF block, is reached. If expression-2 is false, the EXITIF directive is not reached and the statements between the two ENDIF directives are assembled.

REPETITIVE CONTROL

The DO directive initiates and controls repetitive assembly of the statements following it up to the first ENDDO or EXITDO directive.

LABEL	OPERATION	OPERAND
[name]	DO	[expression]

name — Specifies a name for the DO block so that it may be referenced by an EXITDO or NEXTDO directive.

expression — Specifies an integer-valued expression which indicates the number of times the DO block is to be repeated. If expression is omitted, the DO block is repeated until an EXITDO directive is assembled. Forward, external, and address references are not permitted.

EXAMPLE

```
REPEAT   DO   5
```

The group of statements following the DO directive up to the first ENDDO or EXITDO are assembled five times. The label REPEAT may be referenced by an EXITDO or NEXTDO directive.

USAGE NOTES

The following illustrates the simplest form of a DO block:

```
[DO block name]     DO      expression
                    .
                    .       Statements to assemble repetitively.
                    .
                    ENDDO
```

An ENDDO directive must end each DO block.

The Assembler processes each DO block as follows:

1. Establishes an internal counter and defines its value as the value of expression.

2. Evaluates the expression that represents the count.

3. If the count is equal to zero, continues assembly with the statement that follows the ENDDO directive.

4. If the count is greater than zero, processes the DO block as follows:

   A. Decrements the internal counter by 1.

   B. Assembles all statements encountered up to the first ENDDO or NEXTDO directive.

   C. Repeates steps 4A and 4B until the block has been processed the number of times specified by the expression.

   D. Terminates control of the DO block and resumes assembly at the statement following the ENDDO directive.

The DO directive is especially useful for initializing a table. For example, the following DO block initializes a table containing fifty 7-byte records:

```
        DO      50
        DB      23
        DW      ADDR
        DC      'ABCD'
        ENDDO
```

A DO block can be nested within other DO blocks or within an IF block. The following illustrates a DO block nested within a DO block:

```
name    DO      expression
        .
        .
        .
        DO      expression
        .
        .
        .
        ENDDO
        .
        .
        .
        ENDDO
```

When DO loops are nested, the inner DO block must be enclosed within the outer DO block. Control can be transferred outside the DO block at any time by means of an EXITDO, NEXTDO, or EXITIF directive.

IMMEDIATE ITERATION

The NEXTDO directive causes the Assembler to perform immediately the next iteration of the current DO block or a named DO block.

LABEL	OPERATION	OPERAND
	NEXTDO	[name]

name — Specifies the name of the DO block to be immediately iterated. If name is not specified, the current DO block is repeated regardless of whether or not it has a name.

EXAMPLE

```
NEXTDO   LOOP3
```

The enclosing or current DO block named LOOP3 is performed immediately.

USAGE NOTES

One or more NEXTDO directives may be used within a DO block. The NEXTDO directive is useful only in a conditionally assembled piece of code. The following illustrates the placement of a NEXTDO directive within an IF block which is placed inside a DO block:

```
name-1    DO        expression-1
          .
          .
          .
name-2    DO        expression-2
          .
          .
          .
          IF        expression-3
          NEXTDO    name-1
          ENDIF
          ENDDO
          .
          .
          .
          ENDDO
```

EXITING A DO BLOCK

The EXITDO directive causes the Assembler to terminate processing of the statements within the current DO block or a named enclosing DO block.

LABEL	OPERATION	OPERAND
	EXITDO	[name]

name — Specifies the name of the DO block to be immediately terminated. If name is not specified, the current DO block is terminated regardless of whether or not it has a name.

EXAMPLE

```
EXITDO   LOOP3
```

The enclosing or current DO block named LOOP3 is terminated immediately. Assembly resumes following the ENDDO statement which ends the DO block named LOOP3.

USAGE NOTES

One or more EXITDO directives may be used within a DO block. The EXITDO directive is useful only in a conditionally assembled piece of code. The following illustrates the placement of an EXITDO directive within an IF block which is placed inside a DO block:

```
name-1    DO        expression-1
          .
          .
          .
name-2    DO        expression-2
          .
          .
          .
          IF        expression-3
          EXITDO    name-1
          ENDIF
          .
          .
          .
          ENDDO
          .
          .
          .
          ENDDO
```

ENDING A SOURCE PROGRAM

The END directive terminates the assembly of the current source program.

LABEL	OPERATION	OPERAND
	END	[expression]

expression	Specifies an entry point at which program execution begins.

EXAMPLE

END	Terminates assembly of the current source program.
END 3456	Terminates assembly of the current program and begins execution at 3456.

USAGE NOTES

An END directive is required in a source file, but is not permitted in a library file. If a symbolic label name appears as an expression, it automatically becomes global.

MACRO FACILITY

The macro facility allows the programmer to name a sequence of statements, called a macro, and to code only that macro name whenever the sequence is required in a source program. In addition, the programmer can define a macro which names formal parameters that are replaced by actual parameter values coded with the macro invocation.

During the first pass of assembly, macros are expanded. Macro expansion is enhanced by text substitution which allows strings, including the results of string expressions, to be substituted into the program text.

The use of macros produces shorter source programs and reduces actual coding time. During debugging, a single change in the macro is reflected by the Assembler every time the macro is invoked. Once the macro is debugged, an error-free sequence is ensured every time the macro is invoked.

Macros may be defined in the main source program or in a library file. To access a macro defined in a library file, the name of the library file must be specified to the Assembler in response to the prompt "LIBRARY FILE:". Failure to enter a file name means that macros defined in a library file cannot be accessed. If necessary, a macro definition in a library file can be overriden by a macro definition in the main source program, since the main source program is assembled after the library file.

This chapter introduces the capabilities of the macro facility and illustrates how to define a simple macro, invoke a macro, and pass parameters to a macro from the statement that invokes the macro.

TEXT SUBSTITUTION OPERATORS

The text substitution operators of the Assembler allow character strings to be substituted into the source program text. Text substitution is the means by which formal parameters contained within macros are substituted with actual parameter values; however, text substitution is available both inside and outside of macros.

The following sections describe the general process of text substitution.

Simple Variable Substitution

Simple variable substitution is specified by preceding a variable name with an ampersand. For example:

```
&SAMPLE
```

The ampersand directs the Assembler to substitute the previously defined numerical or string value, defined by the variable SAMPLE, into the source text. For example, in the following sequence of statements:

```
SAMPLE      DEFG      'FOO'
            LDA       &SAMPLE
```

the second statement assembles as:

```
            LDA       FOO
```

Variable Name Delimiter

The exclamation mark is used as variable name delimiter in text substitution. If the above instruction were

```
LDB   &SAMPLE001
```

the text substitution routine tries to use 'SAMPLE001' as the variable name to be substituted. To prevent this situation, the variable name delimiter is used as follows:

```
LDB   &SAMPLE!001
```

Now, the variable name to be substituted is 'SAMPLE'.

DEFINING A MACRO

The MACRO directive names a sequence of statements and optionally defines one or more formal parameters which facilitate the passing of actual values into the sequence of statements.

LABEL	OPERATION	OPERAND
name	MACRO	[formal],...

name — Specifies a valid label which names the sequence of statements enclosed by MACRO and ENDM directives.

formal — Specifies 1 to 64 names which are the formal parameters of the macro.

EXAMPLES

```
WORK    MACRO                   Defines the beginning of a macro and assigns it
                                the name WORK.

PAY     MACRO   RATE,HOURS      Defines the beginning of a macro, assigns it the
                                name PAY, and specifies two formal parameters,
                                RATE and HOURS, which are assigned values from
                                the statement that invokes the macro.
```

USAGE NOTES

The MACRO and ENDM directives identify the beginning and end of a sequence of statements called a macro definition or a macro. For example, the following defines a macro and assigns it the name WORK:

```
WORK    MACRO
        statement-1
        statement-2
        .
        .
        .
        statement-n
        ENDM
```

After the macro is defined, the programmer need only code the name of the macro in the operation field whenever the sequence of statements is required in the source program. For example, the simple statement

```
.
.
.
WORK
.
.
.
```

invokes the macro defined above and generates the sequence of statements defined therein.

Macros may be defined in the source file before they are invoked, in a library file, or both. A macro definition in the source file overrides a macro defined in a library file if the names of both macros are identical.

Since it is more common to generate a sequence of similar rather than identical statements, the statements contained within a macro may contain formal-parameter names which are referenced and replaced by actual values. A formal-parameter name must be specified in the operand field of a macro directive in order to be referenced in the body of a macro. The formal parameter name may be referenced within the body of a macro definition by a statement containing the formal-parameter name preceded by an ampersand. When a macro-invocation statement is encountered, the instructions contained within the macro definition are assembled and the formal values specified in the invocation statement are substituted for each ampersand and formal-parameter name. For example, consider the following macro definition:

```
WORK    MACRO    HOURS
        .
        .
        .
        LDB      &HOURS
        .
        .
        .
        ENDM
```

The above macro defines a formal parameter named HOURS in the operand field of the macro directive. The MOV instruction references HOURS, which is preceded by an ampersand. During macro expansion, the characters appearing in the operand field of the invoking statement are substituted everywhere in the macro body where &HOURS appears. For example, the invocation

```
    WORK      OVTIME
```

results in the following expansion:

```
    .
    .
    .
    LDB       OVTIME
    .
    .
    .
```

The formal parameters behave like DEFL symbols; they are local to a specific invocation of the macro. Up to 64 formal parameters may be specified in the operand field of a macro directive and substitution may occur in the label field, operation field, comment field, or any combination of these fields. For example, consider the following macro definition:

```
    SAVEA     MACRO     TYPE,LOC
              &TYPE     &LOC
              ENDM
```

The invocation

```
    SAVEA     STA,VAL
```

results in

```
    STA       VAL
```

The invocation

```
    SAVEA     LDB,STR
```

results in

```
    LDB       STR
```

Macro invocations may be nested or recursive.

For compatibility with Motorola, the MACRO directive may be spelled MACR and the EXITM directive may be spelled MEXIT.

GENERATING UNIQUE LABELS

A macro call may generate statements containing labels. For example, the following definition is used to set the DE registers to the absolute value of the AX register's content:

```
ABSD        MACRO
            TST       D
            BP        <ENDABSD
            NEG       D
ENDABSD     EQU       *
            ENDM
```

The first time ABSD is called, label ENDABSD is defined. Since ENDABSD has been previously defined, a duplicate label definition error occurs the second time ABSD is called. To avoid duplication, a unique label must be generated each time the macro is called. The Assembler provides a special predefined macro parameter (INDX), which is set to a unique five-digit numeric value each time a macro is called. The macro ABSD may now be defined using INDX, as shown here:

```
ABSD        MACRO
            TST       D
            BP        <AB&INDX
            NEG       D
AB&INDX     EQU       *
            ENDM
```

Assuming that this is the only macro defined and that there are exactly two calls made, the first call defines the label AB00001, and the second call defines the label AB00002.

EXITING A MACRO

The EXITM directive causes all statements following it and preceding the closing ENDM directive in the current or specified macro to be ignored.

LABEL	OPERATION	OPERAND
	EXITM	[name]

name — Specifies the name of an outer macro to be exited. If not specified, the current macro is exited.

EXAMPLE

```
EXITM   MYMACRO        Exits a macro named MYMACRO.
```

USAGE NOTES

The EXITM directive may be used at any point within a macro to terminate processing of the current invocation of the macro and any IF or DO blocks currently active within the macro.

For compatibility with Motorola, the EXITM directive may be spelled MEXIT.

ENDING A MACRO

The ENDM directive terminates a macro.

LABEL	OPERATION	OPERAND
	ENDM	

EXAMPLE

```
ENDM                    Terminates the current macro.
```

USAGE NOTES

An ENDM directive must end each macro. Assembly resumes following the ENDM statement.

DEFINING LOCAL IDENTIFIERS

The DEFL directive defines a temporary symbol whose value is known only within the current level of macro expansion.

LABEL	OPERATION	OPERAND
name	DEFL	expression
name	DEFL	'string'

name	Specifies any valid identifier or label.
expression	Specifies an integer valued or character expression. Forward, external, and address references are not allowed.

EXAMPLE

COUNT	DEFL	5	Specifies COUNT as an identifier whose value, 5, is known only within the current level of macro expansion.
REF	DEFL	'CODE'	Specifies REF as an identifier whose value, the character string 'CODE', is known only within the current level of macro expansion.

USAGE NOTES

Local symbols are symbols whose value is known only within macro expansions at the current level of macro expansion and at deeper levels that do not themselves define that symbol locally. If a local symbol is defined locally in a macro, that symbol is not available once the macro is exited. A local symbol of the same name defined in another macro is considered a different symbol.

ASSIGNING A CHANGABLE VALUE TO AN IDENTIFIER

The DEFG directive assigns a changable value to an identifier.

LABEL	OPERATION	OPERAND
name[:]	DEFG	expression

name — Specifies any valid identifier.

expression — Specifies any expression that is an absolute expression. It may not be relocatable, but may contain an expression involving the difference between two relocatable variables in the same section.

EXAMPLE

```
HERE    DEFG  *-SAM   Assigns the value of the current location counter
                     minus the relocatable label SAM to the identifier
                     HERE.

LAST:   DEFG  1000    Assigns the value 1000 to the identifier LAST.

START   DEFG  'abc'   Assigns the string value 'abc' to the identifier
                     START.

        LDA   &START  Refers to the above set-symbol. This is seen by the
                     Assembler as "LDA   abc".
```

USAGE NOTES

The DEFG directive requires a label field since the function of the directive is to define the meaning of the name in the label field. The symbol "name" is assigned the value of expression by the Assembler. Whenever the symbol "name" is encountered subsequently in the assembly, preceded by an ampersand, this value is used.

Unlike an EQUated value, the name defined in an DEFG directive may be redefined.

For compatibilty with Motorola, the DEFG directive may be spelled SET.

ASSIGNING THE LENGTH OF A STRING TO A SET SYMBOL

The LENGTH statement assigns the length of a string to a set symbol.

LABEL	OPERATION	OPERAND
name	LENGTH	'string'

name — Specifies any valid identifier or label.

string — Specifies any valid character string.

EXAMPLE

```
SLEN    LENGTH    'THIS STRING'    Specifies SLEN as an identifier whose
                                   value is 11.
```

USAGE NOTES

If the set symbol has not been previously defined, a new local set symbol will be defined.

ASSIGNING PART OF A STRING TO A SET SYMBOL

The SUBSTR (substring) statement assigns part of a string to a set symbol. The proper syntax is shown below:

LABEL	OPERATION	OPERAND
name	SUBSTR	expa,expb,'string'

name	Specifies any valid identifier or label.
expa	Defines the beginning character position of the substring; the first character is position 1.
expb	Defines the length of the substring. If expb is zero, the substring will begin with the character defined by expa and continue to the end of the string.
string	Specifies any valid character string.

EXAMPLE

```
ABC     SUBSTR   4,8,'THIS STRING'     Specifies ABC as an identifier whose
                                       value is 'S STR'.
```

USAGE NOTES

If the set symbol has not been previously defined, a new local set symbol will be defined.

LISTING CONTROL

The Assembler provides a set of listing control directives which allow the programmer to control the content and appearance of the assembly listing. **Table 8** lists the listing control directives and their functions.

Table 1-8. Listing Control Directives

Directive	Function
EJE	Causes the assembly to skip to the top of a new page.
PRINT	Controls what is or is not printed in the assembly listing.

SKIPPING TO THE TOP OF A PAGE

The EJE directive causes the assembly listing to skip to the top of a new page, and prints the current title.

LABEL	OPERATION	OPERAND
	EJE	

EXAMPLE

EJE	Causes the assembly listing to skip to the top of a new page and to print a heading.

USAGE NOTES

The EJE directive is used to improve the readability of an assembly listing.

The EJE directive does not print on the assembly listing.

For compatibility with Motorola, the EJE directive may be spelled PAGE.

CONTROLLING WHAT IS PRINTED

The PRINT directive controls what is or is not printed in the assembly listing.

LABEL	OPERATION	OPERAND
	PRINT	option

option	Specifies one or more of the following:
ON	Indicates printing of all subsequent source lines.
OFF	Suppresses the printing of all subsequent source lines until a PRINT ON directive is encountered.
GEN	Lists the source text of a macro expansion and the generated object code.
ALL	Prints all source lines including lines skipped due to conditional assembly directives and the conditional assembly directives themselves. Location counter values and object data for skipped lines are not printed. Lines are printed after set symbol and macro parameter substitutions have taken place. This option affects printing and the display listing identically.

EXAMPLE

PRINT ON	Tells the Assembler to print all subsequent source lines until a PRINT OFF directive is encountered.
PRINT OFF	Tells the Assembler to suppress printing of all subsequent source lines until a PRINT ON directive is encountered.

USAGE NOTES

All PRINT directive options are modal in nature; that is, once in effect they remain in effect until a countermanding option in a PRINT command turns them off.

The ON and OFF Options

Use of the ON and OFF options provides a selective capability that enables or disables the printing of a subset of statements contained in a program section. Printing begins or ends with the statement immediately following the PRINT directive that contains the ON or OFF option. For example, consider the following:

```
XYZ     CSECT
        PRINT ON
        statement A
        PRINT OFF
        statement B
        PRINT ON
        statement C
        PRINT OFF
        statement D
```

The printed assembly listing for the above sequence is as follows:

```
        statement A
        statement C
```

If neither option is specified, the Assembler defaults to ON.

The GEN Option

The GEN option causes all source text and object code generated by macro expansion to be printed.

ASSEMBLER MESSAGES

The following is a complete list of the error and informational messages that are generated by the FD Assembler.

** duplicate symbol **

Attempt to define a label or variable that was previously defined. The second definition is ignored..

** error while writing object file **

Terminates Assembler processing.

** expression has more than 1 relocatable factor **

Expressions with relocatable or absolute terms.

** immediate value > 127 or < -128 **

The instruction allows only a signed 8-bit value. The high-order bit is used as the sign. Respecify.

** invalid forward reference **

Attempt was made to reference a symbol that is not yet defined in this assembly, for example:

```
A  EQU  B
```

where B is not yet defined.

** invalid label **

The specified label is mispelled or not present in the current assembly. The label of an ENDP must be previously defined in a PROC directive.

** invalid opcode **

The specified opcode is either mispelled or not allowed by the Assembler.

** invalid operand **

The specified operand is either mispelled or not previously defined in this assembly. Possible invalid or inactive block name on EXITDO, EXITM, EXITIF, or NEXTDO statement.

** missing END **

An END statement must be the last statement in an assembly.

** more unprintable errors **

Occurs if there are more than 10 errors in one line.

** over 255 externals **

The symbol table allows a maximum of 255 external symbols. Terminates Assembler processing.

** relative jump out of range **

The Assembler has computed the displacement to the destination label, however, that displacement is too large to fit into a valid machine instruction. Either an absolute jump must be used or the jump instruction must be moved closer (usually within -128 to +128 bytes) to the target label.

** relative jump to different section **

A relative jump to a label in a different section cannot be permitted because the location of that section is not determined until link time, and the Linker is unable to compute a relative displacement to be inserted into this instruction.

** relative jump to external symbol **

A relative jump to an external symbol cannot be permitted because the location of that external symbol is not determined until link time, and the Linker is unable to compute a relative displacement to be inserted into this instruction.

** symbol table overflow **

There is not sufficient space in memory for:

a. The macro definition table, which contains one entry for each macro which has been defined in the current assembly and

b. The labels, DEFGs, and DEFLs which have been defined in the current assembly.

The user must reduce the number of macro definitions, or labels, or symbols used in the Assembly.

** undefined symbol **

The user has forgotten to define a referenced variable, or the symbol was mispelled.

** different symbol value in pass 2 **

The user has caused the value of the label in the preceding source line to have a different value in pass 2 than it was assigned in pass 1.

** improper section nesting **

Every section must be fully enclosed in another section.

LIST OF MACRO AND CONDITIONAL ASSEMBLER MESSAGES

** ELSE out of place **

** ENDDO out of place **

** ENDIF out of place **

** ENDM out of place **

** EXITM out of place **

** invalid macro parameter name **

** macro definition out of place **

** macro mesting exceeds 127 **

** missing ENDDO **

** missing ENDIF **

** missing ENDM **

** operand longer than 32 chars **

** undefined SET symbol **

DIFFERENCES BETWEEN THE FUTUREDATA AND MOTOROLA ASSEMBLY LANGUAGES

1. The FutureData Assembler does not allow period or underscore characters in labels; however Motorola only allows 6 character labels. The FutureData Assembler does not generate an error when 6809 register names are used as labels.

2. The two Assemblers use different spellings for some expression operators; the different ones are summarized below:

action spelling	FutureData spelling	Motorola
exponent	not available	!
inclusive or	.OR.	!+
exclusive or	.XOR.	!X
logical and	.AND.	!.
shift left	.SHL.	!<
arithmetic right shift	.SHR.	not available
shift right	not available	1>
rotate right	not available	!R
rotate left	not available	!L

3. The FutureData Assembler uses the & character as a text replacement operator, which may be used inside or outside a macro. Motorola uses the backslash character, which is not usable outside a macro.

4. The FutureData Assembler uses &INDX in a macro to generate unique labels; Motorola uses the .nnnn notation.

5. Here is the equivalence between the FutureData and Motorola directives:

Action	FutureData	Motorola
block store	DO n	BSZ n
zeros	DB 0	
	ENDDO	no equivalent
end assembly	END	END
end IF	ENDIF	ENDC
end of marcro	ENDM	ENDM
equate	EQU	EQU
programmer error	no equivalent	FAIL
define byte	DB	FCB
define word	DW	FDB
define character string	DB 'abc'	FCC 'abc' or FCC 3, abc
reserve space	DS expr	RMB expr
set DEFG	SET	
string length	LENGTH	no equivalent
substring	SUBSTR	no equivalent
conditionals	IF 'a='b	IFC 'a,'b
	IF 'a<>'b	IFNC
	IF a=b	IFEQ a,b
	IF a>=b	IFGE a,b
	IF a>b	IFGT a,b
	IF a<=b	IFLE a,b
	IF a<b	IFLT a,b
	IF a<>b	IFNE a,b
exit conditional	EXITIF blkname	no equivalent
else clause	ELSE	no equivalent
elseif	ELSEIF	no equivalent
iteration	DO	no equivalent
next iteration	NEXTDO	no equivalent
exit iteration	EXITDO blkname	no equivalent
marcro definition	MACRO	MACR
set local	DEFL	no equivalent
exit macro	EXITM	MEXIT
assign program name	NAME string	NAM string
options	PRINT xxx	OPT opt1,opt2,...
set location counter	ORG expr	ORG expr
top-of-form	EJE	PAGE
define reg. list	no equivalent	REG list
set DP	no equivalent	SETDP expr
skip blank lines	SPC expr	SPC expr
title string	no equivalent	TTL string
public variable	PUBLIC name	no equivalent
external variable	EXTRN name	no equivalent

LINKER

INTRODUCTION

The Linker combines relocatable program sections (CSECTs) from Assembler- or Compiler-generated relocatable file(s) to form a single executable absolute object file. Address references between sections that were unresolved at assembly or compile time are resolved, and sections are relocated for loading at absolute addresses. In addition to relocatable sections, the Linker also supports an absolute section, the special program section for which no relocation is necessary.

LINKER INPUT

Assembler- or Compiler-generated relocatable object files may be used as input to the Linker. These object files are distinguished by the "R" attribute. Each relocatable object file may contain a maximum of eight relocatable sections and one absolute section. If two or more input files contain the same CSECT name, the Linker will either:

1) Ignore the second through nth definitions of that CSECT.

2) If the "append" mode is in effect, all CSECTs of the same name are concatenated into a single memory area by the Linker.

If two or more input files contain the ASECT directive, memory allocations may overlap. If CSECT directives are not used in an assembly, object code will be placed in the ASECT.

In order to link symbolic name references in object files, the name must be declared public (using the PUBLIC directive) during the assembly or compilation of the module which defines the name and external (using the EXTRN directive) in any modules which refer to the name. For further information on public symbols, refer to sections later in this chapter.

LINKER OUTPUT

The executable absolute object file generated by the Linker may be loaded into memory and executed by the Debugger. In addition, a memory map and reference list may be displayed or printed.

OPTIONS

Four Linker options are available to choose the desired type of input/output operation. Upon initial entry to the Linker, these options are displayed on the display screen. To select the desired options, type the appropriate letters as listed in **Table 2-1.** Options may be selected in any order either from the keyboard or from a command file.

Table 2-1. Linker Options

Option	Function
D	DELETE UNSPECIFIED SECTIONS Links only those sections which are specifically named in the Linker commands and deletes all other sections. If D is not selected, all sections from all input files will be linked, and the name and length of each section in each input file will be displayed. Sections named in input commands are positioned in memory as requested. All unnamed sections are positioned after the last named section.
S	INCLUDE SYMBOL TABLE Writes symbol tables from all input files to the output file. These tables, which relate public symbols to memory locations, are necessary for symbolic debugging. A symbol table may be placed in each relocatable object file by the Assembler or Compiler.
L	LIST TABLES ON CRT Displays the memory map and reference list.
A	APPEND DUPLICATE SECTIONS The normal operation of the Linker is to overlay multiple occurrences of a section. The Linker provides information in the link map when this occurs. This option forces all occurrences of duplicate sections to be concatenated rather than overlaid. This is particularly useful for compiler-generated assembly language programs, which may use section names in this manner.

COMMANDS

The Linker commands are entered to specify the following information:

1) the input file(s);

2) the output file;

3) the listing file (if any);

4) section placement in memory;

5) lists of section names in the order desired;

6) program execution entry point.

An example Linker display is shown in **Figure 2-1.** Options D and A were not selected. The next <RETURN> will cause Linker processing to begin.

```
========== Linker V5.2 ==================================== FutureData ======


SPECIFY LINKER OPTIONS:
 (D) - DELETE UNSPECIFIED SECTIONS
 (S) - INCLUDE SYMBOL TABLE
 (L) - LIST TABLES ON CRT
 (A) - APPEND DUPLICATE SECTIONS
>LS
Input file options:  (select no more than one per file)
 /G - Globals only in symbol table
 /N - No symbol table at all
 /R - Reference only file (no data/symbols)

Input file:
>1:DEMO.R
Section      Length
CODE         0010
DATA         0BB8
Input file:
>
Output file:
>1:DEMO
Listing file or device:
>P:
Linker input:
>#ORG X'1000'
```

Figure 2-1. Example Linker Display

After the "Linker input:" prompt, the user may use the exclamation point character "!" to insert commentary which is ignored by the Linker but aids the user's memory/understanding. If the exclamation point is the first character of the input line, the whole line is ignored by the Linker; an exclamation point following an input command, on the same input line is treated by the Linker as the end-of-line character.

Each Linker command is defined below in the order of use.

Command	Explanation
input file-spec [input file-spec] . . .	After the Linker options have been entered, the Linker will prompt: INPUT FILE. Type the desired filename(s), one filename and <RETURN> per line. The Linker also supports three switches after an input file specification. The input files to the Linker are relocatable modules generated from the assemblers. They consist of text records, relocation information, and symbol tables. There is a symbol table for public symbols and one for local symbols. The three switches are used by the Linker to control the manner in which the input file is processed.
/R	Use the public symbol table of the file to resolve references in other files, but do not include any object records for this file.
/N	Do not include the symbol table of the file if the S option was chosen from the Linker menu. This is useful for creating smaller debugging modules that can be used with the debuggers.
/G	Similar to the /N switch in that it does not include local symbol tables for the file, but it does include the public symbol tables.
<RETURN>	Press <RETURN> after the last input file-spec has been entered.
[output file-spec]	The Linker prompts: Output file. Type the output file specification and press <RETURN>.
[listing file-spec]	The Linker prompts: Listing file. Type the listing file specification and press <RETURN>.
[#ORG absolute addr]	The Linker prompts: Linker input. Specify the absolute address of the first relocatable program section in the list following the #ORG statement.
[sect-a][,sect-b] [sect-c][,sect-d]...	Enter a list of section names. The sections are positioned in memory in the order named, starting at the address in the preceding #ORG or #SEG statement. If there is no preceding #ORG or #SEG directive, the starting address is zero.

Command	Explanation
[#ORG absolute addr] [sect-e]	Additional starting addresses, each followed by a list of sections, may be specified. The absolute address is decimal, hexadecimal, binary, or octal constant as described in Chapter 1. It is strongly recommended that the ORG addresses start at small values and only increase. This makes the memory map output much more readable.
[#SEG [long addr]]	This command specifies a 32-bit hexadecimal segment origin (similar to an ORG address but twice as long). If the long address parameter is missing, the segment origin defaults to the current load location pointer. There may be up to 15 of these commands per link. This directive is used in linking for execution by microprocessors which support more than 64K bytes of address space.
[#END [entry pt name]]	Enter a #END statement to begin Linker processing. entry point name is a public symbol specifying the program entry point. (Section names are automatically public.) If not specified, the program entry point is set to the address of the first section. After processing begins, press <BREAK> to halt Linker processing and view display output. Press <BREAK> again to continue. When processing is complete, the Linker displays the message: Function completed. Press <RETURN>. To begin Linker operation again, enter an input file-spec. To jump to another ADS system program, enter a J command.
[[#DEF] [entry pt name] [public symbol = absolute addr][SEG#]]	This command, followed by symbol definitions, allows the user to define values for external symbols which were not defined previously in some section but are referenced in some section. The DEF command, when used, must replace the #END command. The optional section number is a single hexadecimal digit separated by at least one blank from the address field.

LINKER SUB-COMMAND FILES

The user may invoke sub-command files within the Linker. If the input stream starts with a "." at any point at which the Linker expects input, the remainder of the string is taken as a file specification to be used as an alternate input stream. A sub-command file is distinguished by the "S" attribute. When the end of file marker is reached, the Linker returns to its previous input stream. The sub-command file may be invoked while command file processing is active and may be used any number of times. However, sub-command files cannot be nested.

The example in **Figure 2-1** is repeated in **Figure 2-2**, but in this case a sub-command file, LINK.FILES, is used. The contents of the file are:

```
1:DEMO.R

1:DEMO
P:
```

Figure 2-2 gives the resulting Linker display when run with the sub-command file. The line "Linker input:" is generated from the previous input stream, either via the keyboard or a command file. The lines may not include the command file commands such as n for parameter substitution, L and K.

```
============Linker V5.2====================================FutureData=======

Specify Linker options:
  (D)  Delete unspecified sections
  (S)  Include symbol table
  (L)  List tables on CRT
  (A)  Append duplicate sections
>LS
Input file options:  (select no more than one per file)
 /G - Globals only in symbol table
 /N - No symbol table at all
 /R - Reference only file (no data/symbols)

Input file:
>.1:LINK.FILES
>1:DEMO.R
Output file:
>1:DEMO
Listing file or device:
>P:
Linker input:
>#ORG X'1000'
```

Figure 2-2. Example Linker Display Sub-Command Files

MEMORY MAP OUTPUT

#ORG commands may cause an CSECT to overlap a previously specified CSECT or the ASECT.

An example memory map display is shown in **Figure 2-3.** The memory map lists, for each section:

1) Its start address in memory (if the #ORG commands were entered with their operands in ascending order, the listed memory locations will also be in ascending order);

2) Its name;

3) The name of the file from which the section was taken;

4) Its length in hexadecimal.

```
============ Linker V5.2 ==================================== FutureData ======

Addr          Section        File             Length

1000          PROGRAM        SKELETON.R       059F
159F          IOCODE         EXEC80.R         0028
157C          DEVICE         DEVICE.R         0005
15CC          INCODE         INPUT80.R        03BD
1989          KBCODE         KB80.R           0077
1A00          TVCODE         TV80.R           032D
1D2D          NIO            NET80.R          0565
2292          STACK          SKELETON.R       0064
22F6          DIO            DISK80.R         09CA
2CC0          LPCODE         LP80.R           0360
D600          WADATA         WORK.R           0200
D800          CRT            TV80.R           07D0

============ Linker V5.2 ==================================== FutureData ======

File          Section   Addr      Length
SKELETON.R    STACK     2292      0064
              PROGRAM   1000      059F
```

Figure 2-3. Example Memory Map Output

If a section was deleted (option D), its name will appear in the memory map and the address will be flagged with a "D". If an section is overwritten by another section, the overwritten section will be flagged with an "O" in the memory map.

```
=========== Linker V5.2 ================================== FutureData ======

Addr          Section        File           Length

0080          INTER$         INIT86.R       0034
0000-1        INIT$          INIT86.R       0027
0027-1        INIT$          RARIT86.R      0006 A
002D-1        INIT$          CRT86.R        0005 A
0032-1        INIT$          TERM86.R       0007 A
0039-1        RUNTIME        MATH86.R       04C6
04FF-1        RUNTIME        STRING86.R     0187 A
0686-1        RUNTIME        SET86.R        0080 A
0706-1        RUNTIME        LARIT86.R      015F A
0865-1        RUNTIME        RARIT86.R      0674 A
0ED9-1        RUNTIME        MISC86.R       006B A
0F44-1        RUNTIME        CRT86.R        005B A
0000-2        ROM$           TMP.R          004D
004D-2        ROM$           MATH86.R       00AD A
00FA-2        ROM$           SET86.R        0060 A
015A-2        ROM$           RARIT86.R      0004 A
015E-2        ROM$           PASCALIO.R     001A A
0178-2        ROM$           REALIO.R       0035 A
01AD-2        RAM$           INIT86.R       0038
01E5-2        RAM$           RARIT86.R      0046 A
022B-2        RAM$           PASCALIO.R     0002 A
022D-2        RAM$           CRT86.R        07D2 A
09FF-2        HEAP$          MISC86.R       0002
3FFE-2        STK$           MISC86.R       0000
0000-3        CODE$          TMP.R          010B
09A4-3        PASCALIO       REALIO.R       02B1 A
010B-3        PASCALIO       PASCALIO.R     0899


SEG Address
 00 00000000
 01 1F054FCA
 02 CD000000
 03 041F047E
```

Figure 2-3a. Example Of Segmented Memory Map Output

Note: Every address in this output is followed by a hyphen and the segment number in which it resides.

REFERENCE LIST OUTPUT

An example Linker reference list display is shown in **Figure 2-4.** The display lists, for each relocatable file specified in the link:

1) Its file name;

2) For each section loaded from that file:

   a. Its section name;

   b. Its starting address in hexadecimal;

   c. Its length in hexadecimal.

3) For each public symbol in the section:

   a. The symbolic name;

   b. The memory address corresponding to that name.

```
========== Linker V5.2 ==================================== FutureData ======

File          Section       Addr      Length
SKELETON.R    STACK         2292      0064
              PROGRAM       1000      059F

              Publics
              PROGRAM       1000           STACK     2292

File          Section       Addr      Length
EXEC80.R      IOCODE        159F      0028

              Publics
              IOCODE        159F           IOEXEC    159F

File          Section       Addr      Length
DEVICE.R      DEVICE        157C      0005
```

Figure 2-4. Example Linker Reference List Display

```
========== Linker V5.2 ================================== FutureData ======

File        Section  Addr     Length
INIT86.R    INTER$   0080-0   0034
            INIT$    0000-1   0027
            RAM$     01AD-2   0038

            Publics
            INIT$    0000-0   INTER$   0080-0  L$0      01AD-D  LASTATE  01E3-3
            RAM$     01AD-D

File        Section  Addr     Length
TMP.R       CODE$    0000-3   010B
            ROM$     0000-2   004D

            Publics
            CODE$    0000-0   M$$$$    0000-0  PACON    0000-0  ROM$     0000-0

File        Section  Addr     Length
MATH86.R    RUNTIME  0039-1   04C6
            ROM$     004D-2   00AD

            Publics
            ARCTAN   03FC-C   COS      0166-6  EXP      01A3-3  LN       0357-7
            RUNTIME  0039-9   SIN      0066-6  SQRT     025A-A

File        Section  Addr     Length
STRING86.R  RUNTIME  04FF-1   0187

            Publics
            CMP$0    05C5-5   CMPS$    05A4-4  PA2ST$   063F-F  PACON$   04FF-F
            SBSTR$   0556-6   STR2P$   0609-9  STR2ST$  0663-3  STRCON$  0529-9

File        Section  Addr     Length
SET86.R     RUNTIME  0686-1   0080
            ROM$     00FA-2   0060

            Publics
            FTBL$    011A-A   IN$01    06DE-E  SETBIT$  0686-6  TBL$     013A-A
            TTBL$    00FA-A

File        Section  Addr     Length
LARIT86.R   RUNTIME  0706-1   015F

            Publics
            DDIV$    0758-8   DMOD$    0848-8  DMUL$    0706-6
```

Figure 2-4a. Example Segmented Linker Reference List Display

LIST OF MESSAGES

**DELETED SECTION REFERENCE IN (filename)

An section which is present in an input file, but is not included in the output file, is needed to resolve address references.

(label) **DUPLICATE PUBLIC IN (filename)

The first occurrence of a public symbol is used for address references. All additional definitions are flagged as errors.

NOT A RELOCATABLE FILE

The Linker input file was not created by the Assembler or Compiler or the file's "R" attribute was changed with the Manager. Assign the "R" attribute to the file.

PARM ERR ... RESPECIFY

There is a syntax error in the Linker input or the specified section name was not found.

**RELOCATION ERROR IN (filename),record # xxxxH

Relocation record number xxxx of the input file was not correctly built. First, try to reassemble; else, notify your Service Representative.

**SEQUENCE ERROR IN (filename),record # xxxxH

The records in the named file are not in proper order. Reassemble or recompile.

**SYMBOL TABLE NOT FOUND IN (filename)

The symbol table was not included when the program was assembled or compiled. Reassemble or recompile, or simply ignore.

**TABLE OVERFLOW

The Linker needs more memory space.

(label) **UNRESOLVED REFERENCE IN (filename)

The external reference was not found in the public symbol table.

**PUB REC LNG ERROR IN (filename),record # xxxxH

Public symbol record number xxxx of the input file was not correctly built. First, try to reassemble; else, notify your Service Representative.

**REL REC LNG ERROR IN (filename),record # xxxxH

Relocation record number xxxx of the input file was not correctly built. First, try to reassemble; else, notify your Service Representative.

**EXT REC LNG ERROR IN (filename),record # xxxxH

External symbol record number xxxx of the input file was not correctly built. First, try to reassemble; else, notify your Service Representative.

**SYM REC LNG ERROR IN (filename),record # xxxxH

Record number xxxx of the input file was not correctly built. First, try to reassemble; else, notify your Service Representative.

**REL REC LNG ZERO IN (filename),record # xxxxH

Relocation record number xxxx of the input file was not correctly built. First, try to reassemble; else, notify your Service Representative.

CHAPTER 3

DEBUGGING

INTRODUCTION

The 2300 Series Advanced Development System (ADS) is a software development tool providing users with editing, compiling, assembling, and linking capabilities. Files are maintained by a File Manager and can be stored and accessed on a variety of devices, including floppy disk, hard disk, printer, and Memory Expansion Unit (MEU).

Once the software is developed, users must have a way to test the software on the actual hardware it was designed for and still be able to make modifications and corrections. The target system often does not have the facilities to allow a user to perform those tests. Its input-output structure may not be suited for gathering relevant statistics about the program. Programmers must write time-consuming diagnostic programs into the main program for debugging. These diagnostics may alter the behavior of a real-time system enough so that it fails to perform when the diagnostics are removed. The target system may have no facilities for isolating small sections of code (through trace or breakpoint functions, for example) so that their behavior may be examined. There must be some sort of interface between the ADS and the target system to provide these facilities.

The Slave Emulator Unit is such an interface. It is a hardware device that emulates or imitates the real-time aspects of the target system. It is connected to the target system via a connector that is exactly the same size, with the same number of pins, as the target processor. Users simply remove the target processor from its socket and substitute the Slave Emulator connector.

The Slave Emulator is also connected to the ADS via an RS-232 serial port. The ADS interprets user commands to the Slave Emulator and transmits simple control instructions to the Slave Emulator for execution. The Slave Emulator transmits raw data to the ADS for format and display. The ADS acts as a user interface and a display station. The Slave Emulator acts as a control unit for the target processor.

The Advanced Development System (ADS) coupled with the Slave Emulator Unit provides a powerful system for developing and testing both hardware and software in a microprocessor-based system. The Emulator is based around the actual target processor to be imitated or emulated (for example, the 6809E microprocessor), and a sophisticated control system is imposed over it to provide the facilities required of a software development tool. Only the processor is emulated, allowing the actual hardware of the target system to be used during software development.

When users are ready to test or debug the software, they invoke the Slave Emulator Debugger and its special features.

- The ADS/Slave Emulator combination provides a display which can be split into multiple windows to allow viewing of different areas of memory simultaneously in hex/ASCII or disassembled format (including user labels), or Logic Analyzer trace data. Each window in the display can track a microprocessor register.

- Standard debugging features include symbolic and arithmetic expression evaluation.

- Flexible memory mapping capabilities partition the Emulator memory into user-defined blocks for RAM and ROM simulation. The entire simulation memory is available to the user. Blocks of simulation memory can be write-protected for RAM and ROM simulation. Write-protection is provided in multiples of 256 byte blocks. Mapping functions allow any 8K or 32K block of simulation memory to be mapped anywhere within the microprocessor's full address space, except when using 64K X 8-bit dynamic simulation memory. This memory is designed to simulate memory throughout the address space of typical 8-bit processors.

- 6809E processor interrupts can be invoked via the keyboard for target system interrupt routine testing.

- Help displays are available with a single keystroke, to provide explanations of functions, parameters, and syntax.

- There are four hardware breakpoints (program execution or data breakpoints) with nesting and complex breaking conditions. Conditions include memory address ranges, data values, a halt/snap mode, breakpoint counts and four external lines for breakpoint qualification. The snapshot mode displays the status of the CPU and memory contents at breakpoints by momentarily halting the processor.

- Microprocessor control lines can be selectively enabled or disabled from the target processor under user control. Programs may be executed at full speed or single-stepped. Full-speed execution is a real-time operation up to 2 MHz.

- Up to four Slave Emulators may be attached to one ADS, allowing the user to emulate multiple target systems.

This chapter contains detailed descriptions of each 6809E Slave Emulator command. Individual command descriptions identify additional features.

CRT DISPLAY

Five main elements form the CRT display:

LOGO LINE

There are two kinds of logo lines displayed: the Executive logo line and the Debugger logo line. The Executive logo line appears as follows:

```
======= Emulator Executive VX.X ============================ GenRad DSD =====
```

X.X refers to the software version used. The Emulator Executive logo line is processor-independent.

When the user switches to a particular Slave Emulator Debugger, the logo line displayed becomes processor-dependent and appears as follows:

```
======= XXXXX Emulator VX.X/ROM VX.X ======================= GenRad DSD =====

      PROCESSOR      SOFTWARE     FIRMWARE
       TYPE          VERSION      VERSION
```

For example:

```
======= 6809E Emulator V2.4/ROM V2.4 ======================= GenRad DSD =====
```

If the software version is not compatible with the firmware version, the Slave Emulator will not operate. Memory and register contents will not display any data in most cases. Please refer to **Table 3-1** to determine compatible software and firmware for all released versions of the Slave Emulator.

Table 3-1. Software/Firmware Matches

SOFTWARE	FIRMWARE
V2.4	V2.4
V2.3	V2.2
V2.2	V2.1

The logo line disappears when the user enters the first keystroke.

COMMAND LINE

Commands are input on the top line of the screen. A blinking cursor indicates the position of the next character to be input.

MESSAGE LINE

Target system state changes and certain error conditions are reported on the message line, which is the line below the command line.

A new status message is displayed in double intensity, with no Emulator identification. If there are Emulators daisy-chained together, the status message is displayed in double intensity, if the status applies to the current Emulator, or in reverse video if it applies to any other Emulator on the chain.

An old status message is displayed in normal intensity, with the message centered and surrounded by asterisks. If there are Emulators daisy-chained together, the message appears on the left side of the screen in normal intensity, with Emulator identification.

REGISTERS AND STACK

The next portion of the display shows the current register contents and the top eight words contained in the user stack.

MEMORY DATA

The remainder of the screen can be formatted in one to four separate data windows. One of these windows is defined as "current". A line of reverse video highlighting the current address line designates the current window. Non-current windows display only one address, not the entire line, in reverse video. Depressing <TAB> redefines the current window clockwise on the display. Commands operating on the memory contents are executed at the address indicated in the current window reverse video line.

Users may select any of eight screen maps (numbered 0-7) for viewing simulation or target system memory. They represent combinations of one to four independent windows. Refer to the SCreen map command for details.

Any window may be set to display either disassembled instructions or hexadecimal and ASCII data. If the Slave Logic Analyzer is present, special display modes may be selected to view bus data. See the WIndow mode command for details.

Figure 3-1 shows these elements in a typical 6809E display. The command line is indicated by the cursor. The remainder of the screen is divided into two independent windows below the register data. Window A (top) displays a disassembled program. This is also the current window, as indicated by the window's long reverse video line. Window B (bottom) displays the contents of memory in hexadecimal and ASCII representation. The current address in Window B is indicated by the block in reverse video. The current Emulator status is displayed on the message line, in this case, ****** Halted ******.

```
========= 6809E Emulator V2.4 /ROM 2.4 ================= GenRad DSD ==========

    A  B                        **** Halted ****            User Stk    Hdwr Stk
D   0000    X 0000   Y  0000   DP  0000                    +0  2000    +0  FF20
U   0000    S FFFF   PC FFFF   CC  D7    EFIZVC            +2  1000    +2  0010
                                                           +4  C000    +4  00C0
                                                           +6  4406    +6  0044

Tracking PC
FFFF  FF2000              STU  >2000
0002  10
0003  00C0                NEG  <C0
0005  0044                NEG  <44
0007  0620                ROR  <20
0009  BC28C9              CMPX >28C9
000C  6711                ASR  -0F,X
000E  12                  NOP

FFD0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  | ................ |
FFE0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  | ................ |
FFF0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  | ................ |
0000  20 00 10 00 C0 00 44 06  20 BC 28 C9 67 11 12 18  |  .....D... (.g... |
0010  2C 4C B9 70 0F 18 04 B8  03 84 37 19 2C 80 0E 90  | ,L.p......7.,... |
0020  3F C4 0A F0 D9 EC B2 8B  29 50 4F E1 7D CE 71 68  | ?.......)PO.}.qh |
0030  07 B0 8C 1F 6A 02 0A 8F  A9 0B 14 0A 92 69 13 8C  | ....j........i.. |
0040  16 FB 82 CA 60 24 5B AA  4D 1F 1E 72 A1 00 25 3F  | .....$[.M..r..%; |
```

Figure 3-1. Typical 6809E CRT Display

HELP DISPLAYS

Help displays explain the operation of Slave Emulator commands. Entering <?> or the <HELP> function key on a blank command line produces the general list of commands shown in **Figure 3-2.** Please refer to **Figure 3-7** to determine the location of <HELP>, since the keycaps may not be marked.

Entering <?> or <HELP> on the command line after any individual command is typed produces a help display explaining the specific operation of that command.

switch display to emulator (0-8)	display
screen map (0-7)	<down arrow> (move forward 1 line)
<tab> (assign new current window)	<up arrow> (move back 1 line)
offset base addr =	<right arrow> (move forward 1 byte)
address space	<left arrow> (move back 1 byte)
window mode	+ (move forward 1 page)
edit memory map (Target/Simulation)	- (move back 1 page)
store	find
set register	execute
breakpoint (0-3)	<step>
reset breakpoint (0-3)	restart
clear all breakpoints	halt
qualify trace	interrupt
mode	enable control lines
load file	check for empty PROM
write	program PROM
assign test memory address	get data from PROM
transcribe keystrokes	verify data in PROM
call command file	?
; (display comment)	jump to system component

Figure 3-2. List Of Commands Help Display

COMMAND COMPLETION AND SYNTAX

GENERAL CONVENTIONS

The following general conventions apply to command input:

1. All input can be entered in either upper- or lowercase. Case is significant only when entering character strings for the "Store" and "Find" commands.

2. Keywords are recognized automatically. When the user enters enough of a keyword to uniquely identify it, the Slave Emulator software appends the remaining text of that keyword to the user's input on the command line and positions the cursor after the full text.

3. Keywords include the following:

   Command names
   Breakpoint and Logic Analyzer parameter names
   Breakpoint and Logic Analyzer mode names
   Target address space attributes
   Interrupt type names and related keyword parameters
   Target control line names

4. <TAB> or <CTRL-I> within command input indicates that a default value is to be supplied. The software formats this default on the command line as if the user had entered it and positions the cursor after the default text. Individual commands may take additional action, such as asking for confirmation. <TAB> is ignored where a default value is not meaningful.

5. The separators allowed between multiple fields within a single command are <SPACE> and <,>. Spaces which precede any field are ignored.

6. Command input normally must be terminated by <RETURN> before the specified action occurs. This termination may take the form of a response to a confirmation request, so that in some cases "Y" is also an appropriate termination.

7. Whenever a command's action may be destructive in any way if an error is entered in its text, that command must be confirmed before it is executed.

FILENAME SYNTAX

Filename syntax recognizes a unit specification field consisting of a device type character followed by a unit number and delimited by a colon. Either the device type or the unit number may be omitted, with the corresponding default being "all device types" or "all units". The possible combinations are:

M:	Access MEU (any installed units).
M*:	Access MEU (any installed units).
Mn:	Access MEU units n, where n = 0-3.
D:	Access local floppy disk (any installed units).
D*:	Access local floppy disk (any installed units).
Dn:	Access local floppy disk unit n, where n = 0-3.
*:	Access any unit, either MEU or local floppy.
n:	Access unit n, either MEU or local floppy, where n = 0-3.
P:	Access local printer. (Not applicable for Load command.)

When a filename begins with "P:", the Slave Emulator software immediately asks for confirmation. Other device types require that the user enter the filename string which identifies the file in an MEU or floppy disk directory.

If the user enters more than one unit specification, the last one entered is the one used. The user may not backspace over a unit specification once it has been entered.

There are two ways to enter a filename. The user may type the ASCII string for the filename, followed by <RETURN>. This indicates that the filename is precisely that string. Filename parsing checks to see if the file exists on the selected set of devices, then asks for confirmation in one of the following two ways:

1. If the file exists, the parsing routines prompt "[Old file]". If this name exists on more than one device, a copy on the MEU takes precedence over one on a floppy disk, and one on a low-numbered unit takes precedence over one on a high-numbered unit.

2. If the file does not exist, the parsing routines prompt "[New file]". They create the file on the unit with the most free space among the selected devices, if the user gives confirmation.

The user may also type a string of characters corresponding to the beginning of the filename, followed by a space. If this string matches the beginning of precisely one filename in the selected devices's directories, filename parsing completes the full filename on the command line and asks for confirmation with the prompt "[Old file]". If the string matches no file, or multiple files, filename parsing displays an appropriate message after the last input for several seconds, then looks for further input.

<TAB> completes portions of a filename in the same way that <SPACE> completes the entire filename. <TAB>-induced filename recognition is useful for names of the form <field1>"."<field2>. For example, a directory contains the following files:

```
MODULE.SRC
MODULE.REL
MODULE.OBJ
```

If the user enters a string such as "MOD" followed by <TAB>, filename parsing appends "ULE" and waits for further input, provided no other files begin with "MOD". At this point, typing "SRC"<RETURN>, "S"<SPACE>, or "S"<TAB> selects file "MODULE.SRC".

SLAVE EMULATOR COMMANDS

Slave Emulator commands consist of English language verbs which may be followed by operands. When the user enters enough characters of the verb to uniquely identify it, the Slave Emulator Debugger completes it by appending the remaining characters.

For example, enter the command:

SWitch display to emulator (0-8)

Only the letters SW must be entered to uniquely identify that command. The command characters to be entered by the user are shown in uppercase. They are shown throughout the manual in uppercase to distinguish between user input and system-supplied input. These characters may be entered in either upper-or lowercase. (See SEt dialog modes command for upper-and lowercase options).

Unless otherwise indicated, all commands must be followed by <RETURN> to be entered.

COMMAND LINE EDITING

<BACKSPACE> or <CTRL-H> may be used to delete one command line character at a time. Only that part of the command entered by the operator may be deleted. If a character has already been accepted and the command completed, the character cannot be deleted. <CAN> must be used to cancel the current command.

THE SLAVE EMULATOR EXECUTIVE

The Slave Emulator Executive is the entry point for the Slave Emulation system. To invoke the Slave Emulator Executive, enter the command

JS<RETURN>

on the command line. The display for the Slave Emulator Executive shown in **Figure 3-3** appears on the screen. The status of each Slave Emulator attached to the ADS is displayed. When the Slave Emulator is reset, status is as follows:

1. The target processor displays the message "Initializing", then "Halted".

2. The target system control lines are disabled.

3. The internal memory mapping specifies the entire physical address space mapped to simulation memory.

4. Write access is allowed and simulation memory blocks are mapped to consecutive block addresses starting at location 0.

5. Four commands, SWitch Emulator, SEt Dialog Mode, SPecify screen write options, and Display Comment, are now allowed, as illustrated in **Figure 3-3.**

```
========= Emulator Executive V2.4 ======================== GenRad DSD =========


Emulator 1        6809Ev1    Halted


Emulator Executive commands are:

          Switch display to emulator (0-8)
          Set dialog modes
          Specify screen write options
          ;(display comment)
```

Figure 3-3. Slave Emulator Executive Display

Most of the individual commands in the Executive produces a help display when <?> or <HELP> is entered. The Executive commands are described in the following paragraphs.

SET DIALOG MODES

Users can choose whether to allow upper-and lowercase or to convert all alphabetic characters to uppercase with the SEt Dialog Modes command. The user can also define <RETURN> to signify either YES or NO when the system requires command confirmation. Most Slave Emulator commands require a confirmation to execute.

Upper-and lowercase entry and <RETURN> to confirm means yes are the default values for this command.

This command can only be accessed from the Slave Emulator Executive display.

This command sets modes to control dialog functions.

First select a mode, then specify the value for that mode.

- "k" selects keyboard input mode:

  - "a" specifies "keyboard input allows lower case"

  - "f" specifies "keyboard input folds lower case": All lower case characters will be folded into upper case.

- "r" selects return-to-confirm mode:

  - "y" specifies "return to confirm means yes": Execute the command or subcommand which prompted "[Confirm]".

  - "n" specifies "return to confirm means no": Abort the command or subcommand which prompted "[Confirm]".

Figure 3-4. Set Dialog Modes Help Display

SWITCH DISPLAY TO EMULATOR (0-8)

This command transfers control from the Slave Emulator Executive to the specified Slave Emulator Debugger and displays the state of the target system on the screen. At this point all other Debugging commands are functional. To return to the Emulator Executive, use the command "SWitch display to emulator (0-8)", with "0" as the Emulator number, or <RESET>.

Please note that <RESET> resets all Slave Emulators attached to the ADS.

SPECIFY SCREEN WRITE OPTIONS

Users may write the current screen image into a file in UDOS source format. The file is any output file and may reside on the MEU, local floppy, or local printer.

The filename and the options for filtering special characters out of the data in the screen image must be specified. If the file is being written to the printer, the option "printable characters only" (P) is assumed, and no filtering options are prompted for. Files written with either the "printable characters only" or "no attribute characters" (N) filtering options can be displayed with the UDOS Editor. The UDOS Manager can dump those files written with the "all characters included" (A) option in hexadecimal format.

The command initializes the file so that the first screen image is written at the beginning of the file. After the file has been initialized, using the "F1" function key appends the current screen image to the file. This key is operational only if the user has completed the "specify screen write options" command since last executing the Slave Emulator software.

This command specifies where and how to write a copy of the screen image in response to function key F1.

Respond to the "Filename:" prompt with the name of the file to write.

Respond to the "Filtering option" prompt with the 1st character of:

-Printable characters: Write bytes valued 20H-7FH, filter out special FutureData characters and display attribute characters.

-No attribute characters: Write bytes valued 00H-7FH, filter out display attribute characters.

-All characters: Write all bytes (00H-FFH)

Writing to P: forces the filtering option to "Printable characters".

Figure 3-5. Specify Screen Write Options Help Display

;(DISPLAY COMMENT)

The Display Comment command allows users to make notations in command files. A command code of ";" from the keyboard specifies that further input from the keyboard or command file is to be echoed as usual, but the input is otherwise ignored.

"; (display comment)" echoes up to 79 characters of commentary text on the command line. This text is normally terminated by return.

Figure 3-6. Display Comment Help Display

START-UP PROCEDURE

The Slave Emulator requires the following ADS configuration:

1. The ADS must be a Z80-based system.
2. The ADS must contain 64K of memory in any configuration.
3. The ADS must have the UDOS boot PROM installed on the Z80 CPU. The boot message must read "UDOS Bootstrap", V1.3 or later.
4. The ADS must have a local disk drive or MEU for Slave Emulator software.

The following is a suggested sequence of events for installation and initiation of Slave Emulation. Information regarding hardware set-up is detailed in the **Slave Emulator Hardware Reference Manual (2302-5003-00).**

STEP 1 Remove the MPIO card from the ADS and strap it for Slave Emulator operation. Strapping should be as follows:

2 3 4 6 7 8 11 12 6S for standalone ADS

2 3 4 6 7 8 11 13 6S for Cluster Network ADS

Return the MPIO card to the ADS.

STEP 2 Connect the supplied RS-232 cable from the Slave Emulator, Port 1, to the ADS rear connector panel, Serial 1. If more than one Slave Emulator is being installed on the same ADS, the Slave Emulators must be daisy-chained as follows:

a. Port 2 of Slave Emulator 1 to Port 1 of Slave Emulator 2.

b. Port 2 of Slave Emulator 2 to Port 1 of Slave Emulator 3.

c. Port 2 of Slave Emulator 3 to Port 1 of Slave Emulator 4.

A maximum of four Slave Emulators may be attached to one ADS.

STEP 3 Connect the Emulator probe to the appropriate jacks on the rear panel of the Slave Emulator. **Note that the jacks are keyed to prevent incorrect installation. Do not force the connection.**

STEP 4 Insure that the Slave Emulator power switch is off. Connect the power cord to the Slave Emulator.

STEP 5 Power up the ADS station and disk drive. Insert the Slave Emulator system diskette into any drive. (The Slave Emulator software can be read from any drive, as long as all necessary files are located on the same diskette.)

STEP 6 Load the Slave Emulator Executive and Debugger by typing JS<RETURN>. The disk drive or the MEU accesses the program and overlays. The display on the CRT should resemble the display below:

```
========= Emulator Executive V2.4 ======================== GenRad DSD =========

Emulator Executive commands are:

         Switch display to emulator (0-8)
         Set dialog modes
         Specify screen write options
         ;(display comment)
```

STEP 7 Turn on the Slave Emulator power switch.

STEP 8 After power is applied, the drive containing the Slave Emulator software will be accessed for several seconds, (or for a longer period of time if daisy-chained Emulators are initializing simultaneously), before the following display appears:

```
========= Emulator Executive V2.4 ======================== GenRad DSD =========

Emulator 1          6809Ev1   Initializing

Emulator Executive commands are:

         Switch display to emulator (0-8)
         Set dialog modes
         Specify screen write options
         ;(display comment)
```

STEP 9 Key in SW1<RETURN> to display status for Emulator 1. The Emulator Debugger screen should now be displayed. Refer to **Figure 3-1** for a sample screen display.

SLAVE EMULATOR SYSTEM FILES

The Slave Emulator system diskette contains the following files:

SLAVEM	Resident portion of the Slave Emulator software.
(SENRCOM)	Non-resident command routine.
(SEHELP)	Text of help displays.
(SEAPDM)	Processor-dependent code and tables used by the ADS.
(SESPDM)	Processor-dependent code and tables used by the Slave Emulator.
(SYM1)	Symbol table for Emulator 1.

Note that there is only one symbol table file allocated (SYM1). If there are Slave Emulators daisy-chained together (up to a maximum of four), additional symbol table files, one for each Emulator, must be generated. Use the File Manager to create a file (SYMn) for each Slave Emulator, where n is the Slave Emulator number.

SAMPLE COMMAND SEQUENCE

The following is a sample command sequence to initiate the Slave Emulation process:

JS	Jump to the Slave Emulator Executive. This is the beginning of Slave Emulator operation.
SW1	Switch to emulator (in this case, Emulator 1).
SC	Screen map selection. Provides a selection of eight screen maps for viewing program data.
WI	Window mode command. Data may be displayed in any combination of symbolic or hex format, and may be addressed as absolute or offset. Slave Logic Analyzer displays are also set with this command to display cycle or waveform data.
ED	Edit memory map substitutes Slave Emulator memory for target system memory, or assigns address ranges for Slave Emulator simulation memory.
EN	Enable control lines. This command displays available control line names and allows the user to enable or disable control signals generated by the target system.
L	Load program. This command loads a file from the MEU or a local floppy disk.
B	Set breakpoint. Up to four hardware breakpoints may be set with this command.
EX	Execute the program.
ST	Store the specified expression or string beginning at the current location.
H	Halt. Stop the target system and display the current register values.
CL	Clear all breakpoints.
W	Write to the disk.

COMMANDS

Each command available on the 2302 Slave Emulator system is described in this section of the manual. The commands are organized alphabetically. A complete list of commands is contained in **Table 3-2.** Underlined uppercase letters indicate required user input.

Table 3-2. Slave Emulator Debugger Commands

COMMAND	FUNCTION
ADdress Space	Not applicable to the 6809E processor.
ASsign Test Memory Address	Assign test memory to a fixed location in the target address space.
Breakpoint(0-3)	Set a breakpoint.
CAll Command File	Invoke a command file which functions as a command subroutine.
CHeck for empty PROM	Check for empty PROM.
CLear all breakpoints	Clear all breakpoints.
Display	Position data in current window.
; (Display comment)	Allow user to insert comments in command files.
EDit Memory Map	Specify configurations of simulation memory and target address space.
ENable Control Lines	Enable or disable control lines from target system.

Table 3-2. (Continued)

COMMAND	FUNCTION
EXecute	Begin program execution at the location pointed to by the program counter.
Find	Search for data strings in memory.
Get data from PROM	Move data from PROM to simulation or target memory.
Halt	Stop target processor.
Interrupt	Simulate an interrupt to the target processor.
Jump to system component	Invoke an operating system component or a command file.
Load	Load object file into simulation or target memory.
Mode	Allow user to specify modes for breakpoint and Logic Analyzer operation.
OFfset Base Addr	Define display offset base address (effective only when window mode is set for "Offset Addresses".
Program PROM	Program a PROM.
Qualify trace	Specify Logic Analyzer trace qualification.
RESEt breakpoint(0-3)	Clear a single breakpoint.

Table 3-2. (Concluded)

COMMAND	FUNCTION
RESTart	Restart the target processor by asserting a hardware reset.
SCreen Map	Specify a screen map (number and configuration of windows).
SEt Dialog Modes	Modify uppercase/lowercase, yes/no indicates <RETURN> confirmation. May be used only within the Slave Emulator Executive.
SEt Register	Modify register contents.
SPecify Screen Write Options	Write the current screen into a source file.
STore	Store data into memory.
SWitch Display to Emulator (0-8)	Activate individual Slave Emulator displays.
Transcribe Keystrokes	Causes Slave Emulator software to copy all keystrokes into a source file suitable for later use as a command file.
Verify data in PROM	Compare PROM with simulation or target memory.
WIndow Mode	Set window mode to format in symbolic (disassembled) or hexadecimal format, to display cycle, waveform, or execution data, and to use either offset or absolute addresses.
WRite	Write simulation or target memory contents to an object file.

Table 3-3. Special Keys

SPECIAL KEYS	PURPOSE
<BACKSPACE>	Edit the command line.
<CAN>	Cancel command or sub-command.
<LOAD>	Return to the boot loader.
<RESET>	Reset both the ADS and the Slave Emulator system.
THE FOLLOWING KEYS ARE COMMANDS:	
<TAB>	Redefine the current window or supply default parameters.
<RETURN>	Update display to current status.
<STEP>	Execute program single step.
<?>	Produce help display.
Up, Down, Left, Right Arrows	Position data in current display window.
<+>,<->	Position data in current display window.
<HELP>	Produce help display.
F1 (function key)	Write display function key; causes the current screen image to be appended to the file specified in the last "Specify screen write options" command.
;	Comment command; allows command files to supply prompts to the user.

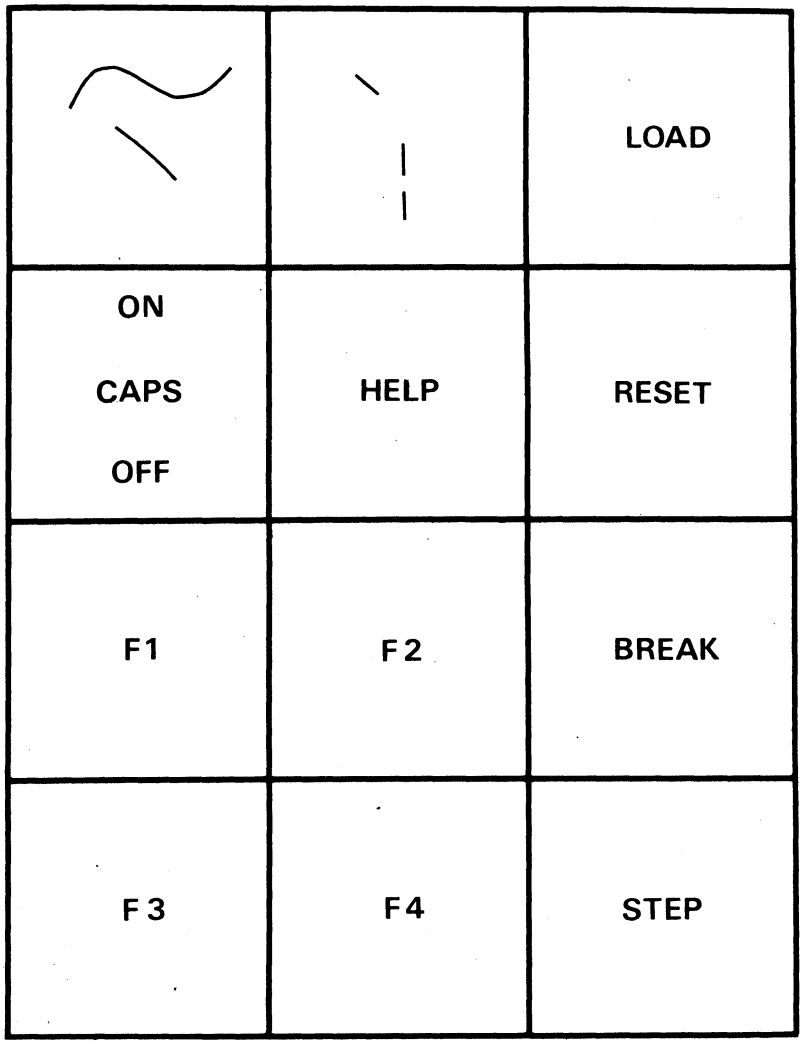


Figure 3-7. Special Keys

ASSIGN TEST MEMORY ADDRESS

ASsign test memory address = expr

FUNCTION:

This command allows users to force test memory to reside at a fixed location of their choice in the target address space.

OPERATION:

The user must input the address at which he wishes test memory to reside and <RETURN>.

ASsign test memory address

ASsign test memory address **expr**

expr defines absolute start address for test memory

---

expr Expression formed with + or - operators, symbols and hex numbers

(symbols have the form "#symbol")

Figure 3-8. Assign Test Memory Help Display

BREAKPOINT COMMAND

Breakpoint (0-3) n<RETURN>
n,

FUNCTION:

The Breakpoint command is used to both arm breakpoint n and to set its parameters. Four hardware breakpoints (0-3) are available.

OPERATION:

n specifies the breakpoint to be armed or to have parameters set.

<RETURN> arms breakpoint n and sets its parameters to defaults: break on instruction execution from current location of current window. To disarm the breakpoint, use the RESEt or CLear commands.

"," specifies that breakpoint parameters are to be set or changed. The "," activates the following subcommands:

OPTION	EXPLANATION
Count = n	1≤n≤32,767. Breakpoint parameters must be satisfied n times before the breakpoint is triggered. If the value entered is not in this range, a message ****Invalid count value**** flashes on the command line and aborts the subcommand. The old count value is not altered.
Address = <abs-addr>	<abs-addr> is any expression representing a 16-bit absolute address. If this expression is a hexadecimal number, "X" may be used in place of individual hexadecimal digits to indicate that any value matches. Breakpoints may break on a value less than or equal to, or greater than or equal to, the specified value. Breaking on an address greater than or equal to the specified value requires that an even address be specified.
Data =<data>	<data> is any expression representing a 16-bit data value. If this expression is a hexadecimal number, "X" may be used in place of individual hexadecimal digits to indicate that any value matches.

OPTION		EXPLANATION
External lines = bin 4		bin 4 (4 binary digits) is the status of the four external probes. X is used for "don't care" bits. The rightmost digit is line E0, the leftmost digit is line E3.
Instruction/data =	I	I specifies break on instruction execution.
	D	D specifies break on data access.
	X	X specifies "don't care".
Read/write =	R	R specifies break on read access.
	W	W specifies break on write access.
	X	X specifies break on "don't care".
Memory/IO =	M	M specifies break on memory address space access.
	I	I specifies break on I/O address space access.
	X	X specifies 'don't care'.
Halt/snap =	H	H specifies that the target system be halted when all breakpoint conditions are satisfied.
	S	S specifies that the display be updated with the target status when breakpoint conditions are satisfied and that the target system resume executing when the update is complete. The S (snap) option stops the emulator from several hundred microseconds to several milliseconds, depending on the target system's clock rate and other timing parameters, except when using scope 0 with "Run after trace" mode set. Refer to Qualify Trace command. The Slave Emulator processes a "halt" regardless of how short the interval is between snapshots. If the user issues commands other than "halt" while the interval between snapshots is extremely short, these commands are queued for processing AFTER the next halt command.

```
Break parameter:

breakpoint (0-3) n  delim                    |   Breakpoint 1
    count = dec                              |        Count = 1
    address = [<= or >=] abs                 |        Address = FFFF
    data = data                              |        Data = XXXX
    external lines = bin4                    |        External lines = XXXX
    instruction/data = I, D, or X            |        Instruction/data = I
    read/write = R, W, or X                  |        Read/write = R
    memory/io = M, I, or X                   |        Memory/io = M
    halt/snap = H or S                       |        Halt/snap = H
                                             |_____________________________

 delim    , to edit parameters, other to set execution breakpoint
n         Breakpoint number:  0-3
 dec      Decimal repetition count:  1 to 32767
 abs      Expression or up to 6 hex digits, X for "don't care" digit
 data     Expression or up to 4 hex digits, X for "don't care" digit
 bin4     4 binary digits, X for "don't care" digit
```

Figure 3-9. Breakpoint Help And Status Display

CALL COMMAND FILE

CAll command file(filename)

FUNCTION:

This command invokes a command file which functions as a command subroutine. It does not require jumping to another system component.

OPERATION:

The Call Command File command first prompts for a filename, which the user enters using the standard Slave Emulator filename dialog conventions. The user is then prompted for parameters; entering only <RETURN> indicates that no parameter values will be supplied to the command file. Otherwise, the input is a sequence of strings separated by commas and terminated by <RETURN>.

Call command file(filename)

This function invokes a command file without jumping out of the Slave Emulator software.

Respond to the "call command file(filename)" prompt with the name of a UDOS source file containing valid Slave Emulator command input.

Respond to the "Parameters" prompt with:

-- <RETURN> if the command file does not require parameters.

-- Text of parameters, terminated by <RETURN>. Multiple parameters are separated by commas.

Figure 3-10. Call Command File Help Display

CHECK FOR EMPTY PROM

CHeck for empty PROM

FUNCTION:

The CHeck command verifies that a PROM is completely erased.

OPERATION:

The command initially prompts for PROM type. The following PROM types are valid: 2704, 2708, 2716, 2516, 2732, 2532, 2758. <TAB> defaults initially to 2716. If the PROM type is changed, the new value becomes the next default.

```
                n
PROM length:
              <TAB>
```

n is an expression that specifies the number of bytes to be programmed beginning at the start of the PROM. n is a hexadecimal value.

<TAB> defaults to program the complete PROM.

```
                          n
Interleaving factor:
                        <RETURN>
```

n specifies the interleaving factor between bytes stored in memory.

Interleaving factor is not meaningful to the Check command. It is prompted for consistency with dialog for the other PROM commands. <TAB> initially defaults to 1. The last value set for any PROM command is the default value.

For further information, please refer to the **EPROM PROGRAMMER USER'S MANUAL, (2300-5035-00).**

```
Check for empty PROM PROM type:

Check for empty PROM
    PROM type:              expr      Type of PROM chip
                            TAB       Default PROM type (initially 2716)

    Length                  expr      Length of data in PROM
                            TAB       Default length:  Chip capacity

    Interleaving factor     expr      Offset between PROM bytes in target
                                      memory

                            TAB       Default interleaving factor:  1

PROM chips supported:
2704, 2708, 2758, 2716, 2516, 2732, 2532
```

```
expr    Expression formed with + or - operators, symbols and hex numbers
        (symbols have the form "#symbol)
```

Figure 3-11. Check Command Help Display

CLEAR BREAKPOINTS COMMAND

CLear all breakpoints [Confirm]

FUNCTION:

This command clears all previously set breakpoints.

OPERATION:

The CLear command disarms all breakpoints. This command disables the hardware breakpoints and initializes all breakpoint parameters to default values. There is no help display for this command.

DISPLAY COMMAND

Display

FUNCTION:

The Display command selects the area of memory to be displayed in the current window. addr may be an absolute hexadecimal address or one of the other forms described below.

OPERATION:

```
Display #symbol
        or
Display #module:symbol
```

symbol is defined in the original assembly language program. Display is positioned to the address associated with that symbol. A "#" character precedes the symbol name.

Symbols may also be qualified by module names by entering the command as "#module:symbol".

Display ±expr

expr is evaluated as a 16-bit value and is added or subtracted to the current display address. The value of expr is saved for subsequent Display ± commands.

An expression consists of one or more terms separated by + or - signs. Each term may be a hexadecimal value or a symbolic reference.

Display @reg

reg is a register name as shown in the Slave Emulator display. This command displays the address pointed to by the contents of the named register in the current window. The address is formed from both the register contents and the contents of the segment register normally associated with that register, if any.

The display now tracks the contents of the register(s) specified. The message "Tracking reg:reg is displayed in the top line of the window.

```
Display @sreg:reg

@sreg:reg defines a segment and an ordinary base register pair when the normal
defaults are not appropriate.

Display BReakpoint(0-3) n

Positions current window to address specified by breakpoint n.  If the
breakpoint address contains "don't care" bits, they are replaced by "0" bits.




Display


                expr            address = expr
                + expr          address = current address + expr
                - expr          address = current address - expr
Display         @ reg           address = contents of reg relocated by default
                                sreg
                @ sreg:reg      address = contents of reg relocated by sreg
                @ sreg          address = 0 relocated by sreg
                Breakpoint n    address = address of breakpoint n

Display +expr or -expr saves expr as page size for + or - commands
```

```
expr      Expression formed with + or - operators, symbols and hex numbers
          (symbols have the form "#symbol")
reg       Any register in the target processor
sreg      Any register in the target processor used to specify
          the base of a memory segment
n         Breakpoint number:  0-3
```

Figure 3-12. Display Command Help Display

EDIT MEMORY MAP COMMAND

EDit memory map (Target/Simulation) [T / S] [Confirm]

FUNCTION:

The Edit Memory Map command allows the user to use all or part of a target system's memory. It defines the emulation environment of the target processor in terms of memory access behavior.

Simulation memory is used in place of target system memory for code generation and testing in the early stages of a project while the hardware is being developed. As hardware becomes available, Simulation memory can be replaced with target system memory, or used as ROM or RAM simulation in the final testing stages of a product.

The memory mapping facility determines whether to use simulation or target memory. The target processor's address space can be divided into several areas, each with different response attributes.

OPERATION:

Edit Target Memory

Initially, the entire address space of the target processor is mapped to direct all memory accesses internally to the Slave Emulator, and to allow write accesses. This is the default and reset condition for this command.

The Target (T) option controls the direction of memory accesses and the attribute of specified areas of the target processor's address space. The command prompts for several parameters and uses these to build a table of descriptions of memory mapped areas. This table is sent to the Slave Emulator for processing when the user completes the command.

Target Option Subcommands:

Block start: addr
[<RETURN>]

addr is the hexadecimal address of a block of Slave Emulator memory to be assigned to the target system. This address is a multiple of X'100'.

If <RETURN> is entered instead of an address, the command is terminated. See **Figure 3-13** for the EDit memory map (Target)help display.

Block length: n

n is the hexadecimal length of the block of memory to be mapped. n must be in multiples of 256 (X'100') bytes.

After the start and length of the block are entered, the command prompts for the attributes of the block. These attributes describe where the memory accesses are to be directed and the types of accesses allowed.

External/Simulation

E directs all memory accesses to the defined block to the target system.

S directs all memory accesses to the defined block to be internal to the Slave Emulator and does not pass them to the target system.

The Write Allow/Prohibit attribute is the final prompt. It controls write accesses to Simulation memory for ROM/PROM simulation. Setting the attribute to Prohibit Write prevents writing to Simulation memory. This memory protection feature is meant to work only in conjunction with Simulation memory. No control over target system memory behavior is possible.

Allows/Prohibits write access

A allows a write to the defined block. P prohibits writes to the defined block.

Figure 3-14 shows the map summary displayed on the screen when using the Target option.

```
Block start:


Target address space map selects attributes for blocks of target addresses.

Command dialog identifies a block by start address and length,
then assigns attributes to it.

    Start address identifies block's first byte, must be multiple of 256.
    --Enter hexadecimal absolute address or <return> to quit.

    Length identifies extent of block, must be multiple of 256.
    --Enter length in hexadecimal.

    Memory attribute (Simulation/External) selects simulation memory in SECU
    or external memory in target system.

    Write protect attribute (Allow/Prohibit) allows or prohibits
    write accesses to this block.
```

Figure 3-13. Edit Memory Map (Target) Help Display

```
Block start:


Attributes:  Simulation/External, Allow/Prohibit writes ("EA" block not shown)
          Start   End   Length  Addr     Start    End   Length  Addr
          0000    FFFF  10000    SA
```

Figure 3-14. Initial Edit Memory Map (Target) Display

Edit Simulation Memory

Upon initialization or reset, the Slave Emulator maps all installed Simulation memory to contiguous addresses starting at location zero in the target processor's address space.

The Edit Simulation Memory command is used to position physical blocks of Simulation memory within the address space of the target processor. These physical blocks are of fixed size, dependent upon the type of memory board installed in the Slave Emulator. This command must be used with the Edit Target Memory command to set up the emulation environment for the target procesor. Care must be taken to first download any code to be executed from this area before attributing it as "Write Prohibit", if the area is to be used in ROM/PROM simulation. Code can not be downloaded into a write-protected area of Simulation memory.

Edit Simulation Memory Subcommands:

```
Block number:   n
              [<RETURN>]
```

n is the block number which identifies each installed block of simulation memory. Blocks are sequence-numbered.

<RETURN> terminates the command. See **Figure 3-15** for the Edit Memory Map (Simulation) help display.

```
Block start:    addr
              [<RETURN>]
```

addr is the absolute (target) address assigned to the first byte of this block of simulation memory. addr must be a multiple of the block length. Blocks of Simulation memory cannot be positioned so as to overlap each other. The command dialog will refuse input that attempts to overlap blocks.

Entering <RETURN> without an address will unmap the block. Unmapping a block of simulation memory prevents its use.

Figure 3-16 shows the map summary displayed on the screen when using the simulation option command.

```
Block number:


Simulation memory map assigns target addresses to blocks of simulation memory.

    Block number identifies each installed block of simulation memory
    --Enter decimal integer to select a block, or <return> to quit.

    Start address is target address at which block's first byte is mapped
    --Enter hexadecimal absolute address, or <return> to unmap block.

Block start address must be multiple of block length and blocks cannot overlap.
This map does not affect whether target system accesses go to simulation
memory; only target address space map selects access to simulation memory.
```

Figure 3-15. Edit Memory Map (Simulation) Help Display

```
Block number:

  Block =   Start     Length    End       Block =   Start     Length    End

    00      00000     01FFF     2000        02      04000     05FFF     2000
    01      02000     03FFF     2000        03      06000     07FFF     2000
```

Figure 3-16. 32K System Initial Edit Memory Map (Simulation) Display

ENABLE CONTROL LINES COMMAND

ENable control lines: X

FUNCTION:

The ENable control lines command determines to which target system's control lines the Slave Emulator will respond.

OPERATION:

X represents one of the following:

NONE	no lines active (default)
ALL	all lines active
RESET	reset line
NMI	non-maskable interrupt
IRQ	interrupt request line
FIRQ	fast interrupt request line
HALT	halt line
TSC	tristate control line
CLK	clock

The user need only enter enough characters to uniquely identify the control line. The system will complete the control line name. The "-" character in front of the control line name disables that line or block of lines. "-ALL" disables all lines. Reverse video indicates active lines. <RETURN> terminates the command.

ENable control line

Enter name from list below to enable control line(s)
Enter -name from list below to disable control line(s)
Enter RETURN to terminate command

none
all
reset
nmi
irq
firq
halt
tsc
clk

Figure 3-17. 6809E Enable Control Lines Display

EXECUTE COMMAND

EXecute

FUNCTION:

The EXecute command begins user program execution.

OPERATION:

The current contents of the Program Counter (PC), as shown in the Register Display, determine the memory address at which execution begins.

Execution is halted by one of the following methods:

Encountering an active breakpoint.
Using the Halt command.
Using the Load command.
Using the STEP key.

EXecute [Confirm]

Execute causes the target system to resume execution.

Execution resumes at the address contained in the target processor's program counter register, relocated by a segmentation register if appropriate.

Figure 3-18. Execute Command Help Display

FIND COMMAND

```
          <data>[<data>]
Find       'string'
           <TAB>
```

FUNCTION:

All forms of the Find command search memory for the occurrence of data or strings.

OPERATION:

<data> is a 1- to 8-digit hexadecimal expression, but only the two low-order digits are significant. Each data value corresponds to an 8-bit byte. 'string' is a series of ASCII characters enclosed in single quotes ('). <TAB> defaults to the operand of the last Find or STore Data command.

The Find command allows the user to enter only 16 bytes of data to the command line. When the user enters the last byte that the command line can accommodate, the Debugger supplies the byte string's delimiter.

Find

```
        expr expr...
Find    'ascii string'
        <TAB>
```

expr expr... is a string of expressions representing byte values

'ascii string' may contain any ascii characters except its delimiter "'"

<TAB> defaults to the operand of the last store or find command

Find begins searching one byte after the current location in the current window;
It searches to the end of:

- --The target address space, if a window shows absolute addresses
- --Window's 64K segment, if window shows offset addresses

Figure 3-19. Find Command Help Display

GET DATA FROM PROM COMMAND

Get data from PROM

FUNCTION:

The Get command loads the information contained in a PROM into the target address space beginning at the current address of the current window. This command may also be used to fill memory (memory is filled with FFFF's) if the PROM programmer module is not connected.

OPERATION:

PROM type:

The following PROM types are valid: 2704, 2708, 2758, 2716, 2516, 2732, 2532. <TAB> initially defaults to 2716. The default is the last PROM type entered.

```
                n
PROM length:
              <TAB>
```

n specifies the length of data to be transferred from PROM. <TAB> defaults to PROM size.

```
                     n
Interleaving factor:
                   <TAB>
```

n specifies the interleaving factor used to move bytes of PROM to memory in the target address space. <TAB> initially defaults to 1. If n = 2, sequential bytes may be programmed alternately in multiple PROMs.

For further information, please refer to the **EPROM PROGRAMMER USER'S MANUAL (2300-5035-00).**

```
Get data from PROM PROM type:

Get data from PROM
    PROM type:              expr      Type of PROM chip
                            TAB       Default PROM type (initially 2716)

    Length                  expr      Length of data in PROM
                            TAB       Default length:  Chip capacity

    Interleaving factor     expr      Offset between PROM bytes in target
                                      memory

                            TAB       Default interleaving factor:  1

PROM chips supported:
2704, 2708, 2758, 2716, 2516, 2732, 2532
    ** This command executes for several minutes for typical PROM types **
```

```
expr     Expression formed with + or - operators, symbols and hex numbers
         (symbols have the form "#symbol)
```

Figure 3-20. Get Data From PROM Help Display

HALT COMMAND

Halt [Confirm]

FUNCTION:

The Halt command suspends execution on the target processor.

OPERATION:

The Halt occurs when the command is confirmed with <RETURN>.

Halt [Confirm]

Halt suspends execution on the target processor

Figure 3-21. Halt Command Help Display

INTERRUPT COMMAND

Interrupt

FUNCTION:

The Interrupt command simulates an interrupt to the target system.

OPERATION:

Interrupts available on the 6809E Slave Emulator are:

NMI	Non-maskable interrupt
FIRQ	Fast interrupt request
IRQ	Interrupt request
SWI	Software interrupt
SWI2	Software interrupt 2
SWI3	Software interrupt 3

The interrupt is issued when <RETURN> is entered.

The user must enter only enough characters to uniquely identify the desired interrupt. For this command, entry of the first character is sufficient.

```
Interrupt

        NMI
        FIRQ
        IRQ
        SWI
        SWI2
        SWI3
```

Figure 3-22. Interrupt Command Help Display

JUMP TO SYSTEM COMPONENT

J character(s)

FUNCTION:

"character(s)" is the first character or characters (up to a maximum of 10) of a Z-attributed system file name. Entering "JS" loads the Slave Emulator Executive from the bootstrap or from another system program, such as the Manager or Editor.

OPERATION:

The Jump to System Component command loads an executable file (a Z-attributed file) to the system. Examples of system component files are the Manager, the Editor, the Linker, the Assembler, and the Slave Emulator Executive. Typing a "J" plus the first letter of the system filename loads the file.

For further information, please refer to the **UDOS REFERENCE MANUAL, Chapter 2, UDOS Overview (2301-5002).**

LOAD FILE

Load file filename

FUNCTION:

The Load file command halts the target processor, if running, searches the available storage devices for the specified filename, and loads the file into the target processor address space.

OPERATION:

If the filename is not prefixed by a device type code or unit number, the Slave Emulator software searches for the file first on the MEU, then on all locally installed floppy disk drives, beginning with drive 0.

The Load command issues the following prompt if the file being loaded is in Linker I format:

```
                          addr
Load segment base addr =
                         <TAB>
```

addr specifies the 20-bit absolute start address of the 64K segment into which to load the file.

<TAB> or <RETURN> defaults to 0 initially or to the previous base address. This parameter does not relocate address references within the loader file. Only the Linker performs relocation.

This command automatically distinguishes between Linker I and Linker II files by the contents of the first record. Linker II files require entry of the filename only. Information on segmentation and address space use is contained within the object file itself.

The Load command sets the CS and IP registers to the start address specified by the Assembler/Linker or the Write command, whichever created that file. If the filename specified exists on more than storage device and the user does not explicitly specify a unit or drive number, the MEU and then the floppy disk drives starting with unit 0 will be searched and the first occurrence of that filename will be loaded.

If the specified filename is not found, the message ****no such file**** will be briefly displayed and the system will await the next command.

```
Load file

-------------------------------------------------------------------------------

Load file filename loads the named file into the target address space.

Additional input for Linker I object files:

"address space = same as current window's [Confirm]" prompt appears if
the target processor supports multiple address spaces.  Confirm to load into
the current window's address space, or abort and use "address space" command
to switch address spaces.

The "Load segment base address = expr" subcommand specifies the absolute base
address for a 64K segment which the file will be loaded into.
<TAB> defaults to last value used (0 initially).

-------------------------------------------------------------------------------

filename        Name of an object file
expr            Expression formed with + or - operators, symbols and hex numbers
                (symbols have the form "#symbol")
```

Figure 3-23. Load File Help Display

MODE COMMAND

Mode parameter:

FUNCTION:

The Mode command controls the Slave Logic Analyzer and breakpoint operation. This command's dialog is similar to that of the Breakpoint command and its display appears as shown below when all modes are set to their initial defaults.

OPERATION:

Typing the command automatically brings up the help display shown in **Figure 3-24.**

AND/OR breaks mode specifies how different breakpoints interact:

The "OR" setting indicates that all enabled breakpoints function independently.

The "AND" setting indicates that all enabled breakpoints function together. Initially, only the lowest-numbered active breakpoint is armed; when its conditions are satisfied, it arms the next higher-numbered breakpoint, rather than causing an actual break. The break finally occurs when the highest-numbered active breakpoint's conditions are satisfied.

Snapshot scope indicates how much data the Slave Emulator updates when the target system reaches a snapshot breakpoint.

0:	Nothing	The Slave Emulator does not interrupt the target system and does not report register or memory contents to the ADS. This is useful only when the breakpoints are used to trigger the Logic Analyzer while the target system is running in full-speed emulation.
1:	Register Contents	The Slave Emulator interrupts the target system for about 10 milliseconds to examine the current register contents and reports these to the ADS.

2:	Register Contents and Memory Displayed	The Slave Emulator interrupts the target system for about 100 milliseconds to examine the current register contents and check for altered data in memory areas whose contents are buffered in the ADS for the Debugger's display. Target processor execution restarts as soon as it has fetched new register and memory contents. If a previous set of values is queued for transmission to the ADS, but transmission has not started, the Slave Emulator discards the earlier register or memory contents and reports the most recent data.
3:	Snap All at Each Break	The target processor remains halted until all data reported by the snapshot has been transmitted to the ADS and the ADS has completely updated its display. This ensures that the display always shows consistent snapshot data and that each snapshot's results are displayed. Target processor execution is suspended for a period of time between 1/2 second to several seconds during data transmission and display updating.

In the display of current mode settings, this parameter appears as follows:

0 (Snap nothing)
1 (Snap regs only)
2 (Snap regs & mem)
3 (Snap all)

Enable/Disable Analyzer enables or disables Logic Analyzer operation. In this mode, reaching a breakpoint (either ORed or ANDed) supplies a Logic Analyzer trace trigger. Execution continues until the number of qualified cycles indicated by the trace qualifier's post-trigger delay have been traced.

When enabling the Logic Analyzer the Debugger issues the following information message on the message line:

****** Analyzer enabled, break 3 disabled ******

When disabling the Logic Analyzer the Debugger issues the following information message on the message line:

****** Analyzer disabled, break 3 enabled ******

The Slave Emulator hardware uses breakpoint 3 to specify the trace qualifier when the Logic Analyzer is enabled.

Run/Pause after trace specifies whether or not the target processor is to continue running while the Slave Emulator copies out the contents of the Logic Analyzer's trace buffer. This applies only when the triggering breakpoint specifies the "snap" option.

The Run setting allows the target processor to continue execution. If the snapshot scope mode is 0 this causes Logic Analyzer trace updates with absolutely no interruption of target system execution. Other settings of the snapshot scope mode delay the target system by the times quoted above.

The penalty for this mode is that breakpoints and Logic Analyzer tracing must remain disabled while the Slave Emulator copies the trace buffer, so that the target system may miss breakpoints. The duration of this critical period will be about 5-10 milliseconds.

The pause setting halts the target system while the Slave Emulator copies the Logic Analyzer trace buffer. This ensures that breakpoints cannot be missed and that all target system execution is traced, but it suspends target execution for about 10 milliseconds more than the delay needed to satisfy the breakpoint scope mode.

The Timer Units mode indicates what units the Logic Analyzer's event timer uses. The event timer is active only when breakpoint 2 specifies a snapshot and the following modes are in effect:

```
AND/OR breaks = A
Scope = 0 (snap nothing)
Enable/disable analyzer = E
Run/pause after trace = R
```

Under these conditions the event timer begins clocking real time or bus cycles when the conditions for breakpoint 1 are satisfied, and stops when breakpoint 2 triggers the Logic Analyzer. Breakpoint 0 has no effect on the event timer.

The event timer uses a 24-bit counter in one of three modes:

Microseconds:	Counter increments once per microsecond, elapsed interval is reported to user in microseconds.
Nanoseconds:	Counter increments once per 100 nanoseconds, elapsed interval is reported to user in nanoseconds.
Bus cycles:	Counter increments once per target system bus cycle, elapsed interval is reported to user as bus cycle count.

The elapsed time or bus cycle counts observed by the event timer are reported in the ADS display's message area with the following message:

Running, elapsed time n units

In this message 'n' is the decimal event timer value and ' units ' is 'nanoseconds', 'microseconds', or 'bus cycles'.

"Default all mode parameters to initial values" sets all parameters to the standard value in effect when the Slave Emulator is reset. These settings are:

```
AND/OR breaks = 0
Scope = 3 (Snap regs only)
Enable/disable analyzer = D
Run/pause after trace =P
Timer units = microseconds
```

The ADS requires the user to confirm in order to accomplish this mode change.

"Set all mode parameters for event timing" sets the parameters to the values needed to activate the event timer, and sets the timer units parameter to microseconds. The user must confirm by depressing <RETURN> or "Y" to accomplish the mode change.

Mode parameter:

```
Mode                                  |Breakpoint/logic analyzer modes
                                      |
    and/or breaks = A or 0            |    And/or breaks = 0
    scope =  scope                    |    Scope = 3 (snap all at each break)
    enable/disable analyzer = E or D  |    Enable/disable analyzer = D
    run/pause after trace = R or P    |    Run/pause after trace = P
    timer units =  unit-code          |    Timer units = microseconds
                                      ------------------------------------
    Default all mode parameters to inital values.
    Set all mode parameters for event timing.
```

```
 scope        Scope of information updated at a snapshot breakpoint:
                   0 = Nothing
                   1 = Register contents
                   2 = Register contents and memory being displayed
                   3 = Registers and memory @ every snap
      **Logic Analyzer trace is independent of snapshot breakpoint scope
 unit-code    Units used by event timer:
                   Microseconds
                   Nanoseconds (minimum resolution = 100 nanoseconds)
                   Bus cycles
```

Figure 3-24. Mode Command Help Display

OFFSET BASE ADDRESS COMMAND

OFfset base addr= expr

FUNCTION:

The Offset Base Address command assigns the specified value to the window's absolute base address.

OPERATION:

When the window mode is set for offset addressing, all addresses are treated as displacements from this base address with wraparound at 64K beyond the base address.

expr is an expression, evaluated as a 16-bit absolute base address.

Addresses shown in the display window are relative to the offset specified when window mode is 'offset addresses'. This function can be used to view a particular relocatable program segment (RSEG) relative to its start address, rather than relative to zero in memory. For example, if an RSEG is loaded at location X'1420', the user can enter "OFfset base addr = 1420". Then, by entering "Display 0", the current window shows location X'1420' absolute, but identifies it as location X'0'. In this way, addresses displayed on the screen will match the assembly listing.

OFfset base addr =

Offset base addr = expr

expr defines absolute start address of 64K segment to display
when window mode is set for offset addresses

---

expr Expression formed with + or - operators, symbols and hex numbers
(symbols have the form "#symbol")

Figure 3-25. Offset Base Address Help Display

PROGRAM PROM COMMAND

Program PROM

FUNCTION:

The Program command begins the PROM programming process using data beginning at the current address in the current window. <RETURN> continues the command.

OPERATION:

PROM type:

The following PROM types are valid: 2704, 2708, 2716, 2516, 2732, 2532, 2758. <TAB> initially defaults to 2716. The default PROM type is the last PROM type set by any PROM command.

```
                n
PROM length:
              <TAB>
```

n specifies the number of bytes to be programmed beginning at the start of the PROM. <TAB> defaults to program the complete PROM.

```
                     n
Interleaving factor:
                   <TAB>
```

n specifies the interleaving factor between bytes programmed from the target address space into the PROM. For example, this factor allows alternate bytes to be programmed into 8-bit PROMs to form 16-bit words. <TAB< initially defaults to 1.

For further information, please refer to the **EPROM PROGRAMMER USER'S MANUAL (2300-5035-00).**

```
Program PROM PROM type:

Program PROM
    PROM type:              expr      Type of PROM chip
                            TAB       Default PROM type (initially 2716)

    Length                  expr      Length of data in PROM
                            TAB       Default length:  Chip capacity

    Interleaving factor     expr      Offset between PROM bytes in target
                                      memory

                            TAB       Default interleaving factor:  1

PROM chips supported:
2704, 2708, 2758, 2716, 2516, 2732, 2532
    ** This command executes for several minutes for typical PROM types **
```

```
expr     Expression formed with + or - operators, symbols and hex numbers
         (symbols have the form "#symbol)
```

Figure 3-26. Program PROM Help Display

QUALIFY TRACE

Qualify trace

FUNCTION:

The Qualify Trace command sets parameters to select bus cycles to be traced when the Analyzer is enabled. These parameters are identical to those used with breakpoints, but the meaning of a match condition is "trace this cycle" rather than "break execution". This command also sets the post-trigger delay, which indicates how many bus cycles to trace after the triggering event occurs.

OPERATION:

Qualify trace functions exactly as the breakpoint command, except that the parameters it sets are those of the Logic Analyzer's trace qualifiers (Breakpoint 3). The following display appears on the screen when the command is entered.

```
                    ****Logic analyzer disabled ****
-------------------------------------------------------------------------------
                                        |
qualify trace                           |Trace qualifier
    post-trigger delay =  dec           |    Post-trigger delay = 1
    address =  [<= or >=] abs           |    Address = XXXX
    data =  data                        |    Data = XX
    external lines =  bin4              |    External lines = XXXX
    instruction/data = I, D, or X       |    Instruction/data = X
    read/write = R, W, or X             |    Read/write = X
    memory/io = M, I, or X              |    Memory/io = X
                                        |
-------------------------------------------------------------------------------

 dec          Decimal bus cycle count:  1-255
 abs          Expression of up to 6 hex digits, X for "don't care" digit.
 data         Expression of up to 4 hex digits, X for "don't care" digit.
 bin4         4 binary digits, X for "don't care" digit.
```

Figure 3-27. Qualify Trace Help Display

Trace qualifier parameters are:

Post-trigger delay:	Number of bus cycles to trace after a trigger (after a breakpoint). This is a decimal number 1-255, with the default being 1.
Address:	Address pattern to trace. This is a hexadecimal number in which "X" in any digit indicates "don't care". The number of digits in this number depends on the width of the target processor's address bus. The default is entirely "don't care" digits.
Data:	Data pattern to trace. This is a hexadecimal number in which "X" in any digit indicates "don't care". The number of digits in this number depends on the width of the target processor's data bus. The default is entirely "don't care" digits.
External lines:	External line values to trace. This number consists of four binary digits in which "X" in any digit indicates "don't care". The default is entirely "don't care" digits.
Instruction/data:	
I	Trace only opcode or instruction-fetch cycles.
D	Trace only data access cycles. Most processors do not distinguish between operand fetches and fetches for the second and subsequent bytes or words of instructions.
X	Default; means trace both instruction fetches and data accesses.
Read/write:	
R	Trace only read cycles.
W	Trace only write cycles.
X	Trace both read and write cycles.

Memory/IO:

M	Trace only cycles which access memory.
I	Trace only cycles which access I/O ports.
X	Default; trace both memory and I/O accesses.

The trace qualifier enables tracing of all bus cycles when all of its parameters have the default "don't care" setting.

For further information regarding the Slave Logic Analyzer, please refer to the **SLAVE LOGIC ANALYZER MANUAL (2302-5012).**

RESET BREAKPOINT COMMAND

RESEt breakpoint (0-3) n

FUNCTION:

This command is used to reset a previously set breakpoint.

OPERATION:

This command disables the specified hardware breakpoint.

There is no help display for this command.

RESTART COMMAND

RESTart

FUNCTION:

The RESTart command issues a hardware reset to the Slave Emulator.

OPERATION:

The reset is issued when <RETURN> key is entered. The message "Running" is displayed on the message line.

RESTart [Confirm]

Restart performs a hardware restart of the target system
by applying a reset control signal

Figure 3-28. Restart Command Help Display

SCREEN MAP COMMAND

SCreen map(0-7) n

FUNCTION:

The Screen Map command maps up to four display windows onto the screen, with windows assigned to quadrants of the display as shown below.

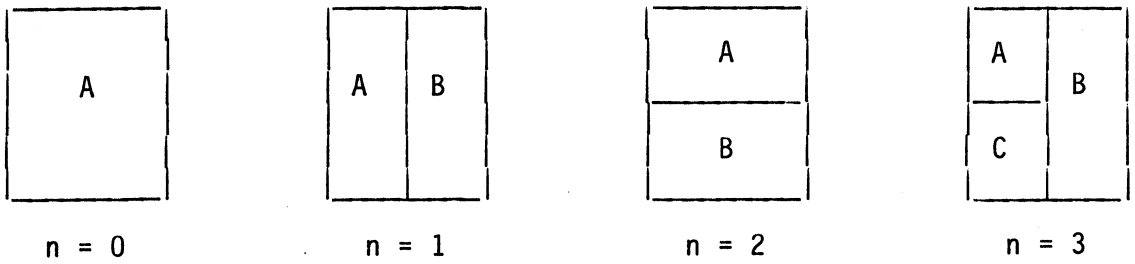
OPERATION:

Any of eight screen maps may be selected for viewing emulator memory. The desired map number is specified by the letter n, as shown in the help display (**Figure 3-29**). The default screen map is n = 2. Each window may display symbolic (disassembled), hexadecimal/ASCII memory data, or Logic Analyzer trace data in any of three formats.

Logic Analyzer trace data can only appear in a window which spans the full width of the screen. (See WIndow mode command). If the current window is the incorrect size to support Logic Analyzer data, an error message stating "**XXX Requires wide window**" will be displayed and the command aborted. This error message may be issued by both the Screen Map command and the Window Mode command.

SCreen map (0-7)

Screen map n maps up to four display windows onto the screen, with windows assigned to quadrants of the display as shown below.
Windows are labelled A - D.



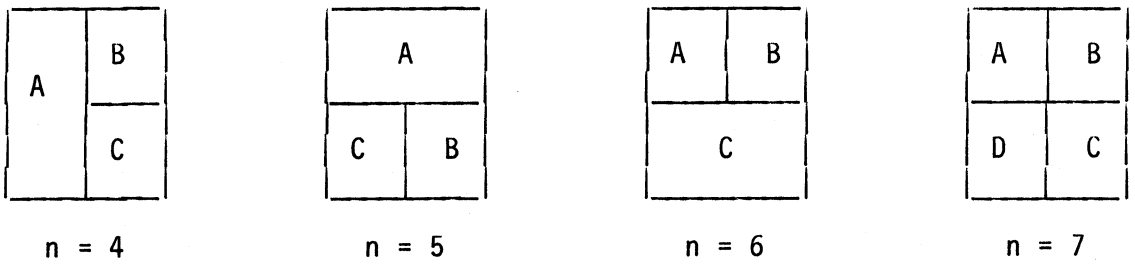


Figure 3-29. Screen Map Help Display

SET REGISTER COMMAND

SEt register = <reg>

FUNCTION:

The SEt Register command sets the contents of the specified register to 'data'.

OPERATION:

<reg> may be any 6809E register name shown at the top of the Debugger display. <data> is one to four hexadecimal digits; a symbol or an expression. The number of significant bits depends upon the size of the register.

The SEt register command also allows the user to change the the target processor's flag or condition code values as if these data were a register.

The 6809E emulator displays the target registers in the following formats:

```
========= 6809E Emulator V2.4 /ROM 2.4 ================== GenRad DSD ==========

    A  B                        **** Halted ****              User Stk    Hdwr Stk
D   0000    X 0000   Y  0000   DP  0000                      +0  2000    +0  FF20
U   0000    S FFFF   PC FFFF   CC  D7    EFIZVC              +2  1000    +2  0010
                                                            +4  C000    +4  00C0
                                                            +6  4406    +6  0044
```

Figure 3-30. 6809E Register Display

```
SEt register

-------------------------------------------------------------------------------

Set register reg = expr

-------------------------------------------------------------------------------

reg       Any register in the target processor
expr      Expression formed with + or - operators, symbols and hex numbers
          (symbols have the form "#symbol)
```

Figure 3-31. Set Register Help Display

STORE COMMAND

```
     <data>[<data>]...
STore <'string'>
      <TAB>
```

FUNCTION:

All forms of the STore command store data strings into memory.

OPERATION:

<data> is a 1- to 8-digit hexadecimal number (or symbol or expression). Only the low 8 bits of this value are significant. <'string'> is a series of ASCII characters enclosed in single quotes ('). <TAB> defaults to the previous <data> or <string> defined in the last Find or STore command.

The STore command does not allow the user to enter enough data to overflow the command line. When the user enters the last byte the line can accommodate, the Debugger supplies input of the byte string's delimiter. Confirmation is requested before entering.

The STore operation begins placing the data or string into memory at the current location in the current window.

```
              expr expr ...
Store         'ascii string'
              TAB
```

expr expr ... is a string of expressions representing byte values

'ascii string' may contain any ascii character except its delimiter, "'", and preemptive keystrokes, such as "?", <BACKSPACE>, <CAN>, etc.

TAB defaults to the operand of the last store or find command

---

expr — Expression formed with + or - operators, symbols and hex numbers (symbols have the form "#symbol")

Figure 3-32. Store Command Help Display

SWITCH DISPLAY TO EMULATOR (0-8)

SWitch display to emulator (0-8) n

FUNCTION:

The SWitch command switches from the Emulator Executive to the specified Slave Emulator and displays its status on the screen.

OPERATION:

If n = 0, control is returned to the Slave Emulator Executive. If n = any number from 1 to 8 inclusive, the specified Slave Emulator is switched to for interrogation.

All communication links are always active. The SWitch command switches control of the ADS display and keyboard to the process communicating with the specified emulator. All emulators may be simultaneously running a user program in the target system.

The control lines, memory maps and execution parameters are unaffected when switching from one emulator to another. **Figure 3-34** displays the status of the Slave Emulator after using the SWitch command. The emulator status may be any one of the following:

```
Initializing
Halted
Running
Completed single step
Stopped @ breakpoint n
Stopped @ ANDed breakpoint
Running, snapshot @ breakpoint n
Running, snapshot @ ANDed breakpoint
Bus timeout
Target system check
```

```
Switch display to emulator (0-8) n

        n selects one of the attached emulators;
        0 returns control to the Emulator Executive
```

Figure 3-33. Switch Command Help Display

```
========= 6809E Emulator V2.4 /ROM 2.4 ================= GenRad DSD ==========

    A  B                        **** Halted ****          User Stk    Hdwr Stk
D   0000    X 0000   Y  0000   DP  0000                  +0  2000    +0  FF20
U   0000    S FFFF   PC FFFF   CC  D7    EFIZVC          +2  1000    +2  0010
                                                         +4  C000    +4  00C0
                                                         +6  4406    +6  0044

Tracking PC
FFFF  FF2000            STU  >2000
0002  10
0003  00C0              NEG  <C0
0005  0044              NEG  <44
0007  0620              ROR  <20
0009  BC28C9            CMPX >28C9
000C  6711              ASR  -0F,X
000E  12                NOP

FFD0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  | ................ |
FFE0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  | ................ |
FFF0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  | ................ |
0000  20 00 10 00 C0 00 44 06  20 BC 28 C9 67 11 12 18  |  .....D... (.g... |
0010  2C 4C B9 70 0F 18 04 B8  03 84 37 19 2C 80 0E 90  | ,L.p......7.,... |
0020  3F C4 0A F0 D9 EC B2 8B  29 50 4F E1 7D CE 71 68  | ?.......)PO.}.qh |
0030  07 B0 8C 1F 6A 02 0A 8F  A9 0B 14 0A 92 69 13 8C  | ....j........i.. |
0040  16 FB 82 CA 60 24 5B AA  4D 1F 1E 72 A1 00 25 3F  | .....$[.M..r..%; |
```

Figure 3-34. Typical Display After Switch Command

TRANSCRIBE KEYSTROKES

Transcribe keystrokes to filename:

FUNCTION:

This command copies all keystrokes into a source file for later use as a command file.

OPERATION:

When the user enters and confirms his filename selection, the ADS opens this file for write access. The end of file pointer is set to show that the file is empty.

The ADS then prompts:

Include prefix sequence? [Confirm]

If the user enters <RETURN> or "Y" to confirm this option, the ADS writes the following text at the beginning of the file:

```
JS
^L
SW1Y
```

This is a standard prefix for command files to be invoked by a Jump command. It implements the following three functions:

1. Jump to the Slave Emulator software. A side affect of this is that it resets all emulators attached to the ADS.

2. Wait for the user to enter <RETURN>. This ensures that the next command can operate properly if Emulator 1 is slow in resetting.

3. Switch to Emulator 1.

Command files to be invoked by the "Call command file" command would not use this prefix.

Transcription of keystrokes begins when the dialog is complete. It ends when the user enters <CTRL-E> or jumps to a different system component.

<CTRL-E> must be entered to ensure that the file is closed properly.

Transcribe keystrokes to filename:

This command initiates transcription of keystrokes to a UDOS source file, which may be used later as a command file.

Respond to the "Transcribe keystrokes to filename:" prompt with the name of the file to write.

Respond to the "Include prefix sequence? [Confirm]" prompt with <RETURN> or "Y" to include the prefix sequence, "N" or <CAN> to exclude it. This prefix sequence, appears as follows in the transcript file:

```
JS
^L
SW1
```

This sequence is used in command files to initiate Slave Emulator operation.

Entering <CTRL-E> at any time terminates keystroke transcription and closes the transcript file.

Figure 3-35. Transcribe Keystrokes Help Display

VERIFY DATA IN PROM COMMAND

Verify data in PROM

FUNCTION:

The Verify command compares the information contained in a PROM with that beginning at the current address of the current window in the target address space.

OPERATION:

PROM type:

The following PROM types are valid: 2704, 2708, 2716, 2516, 2732, 2532, 2758. <TAB> initially defaults to 2716. The default value is the last PROM type entered for any PROM command.

```
                n
PROM length:
              <TAB>
```

n specifies the number of bytes that were verified beginning at the start of the PROM. <TAB> defaults to verify the complete PROM.

```
                     n
Interleaving factor:
                   <TAB>
```

n specifies the interleaving factor between consecutive bytes in the target address space. If n = 2, sequential bytes may be programmed alternately in multiple PROMs. <TAB> initially defaults to 1. The default value is the last value entered.

For further information, please refer to the **EPROM PROGRAMMER USER'S MANUAL (2300-5035-00).**

```
Verify data in PROM PROM type:

Verify data in PROM
    PROM type:              expr      Type of PROM chip
                            TAB       Default PROM type (initially 2716)

    Length                  expr      Length of data in PROM
                            TAB       Default length:  Chip capacity

    Interleaving factor     expr      Offset between PROM bytes in target
                                      memory

                            TAB       Default interleaving factor:  1

PROM chips supported:
2704, 2708, 2758, 2716, 2516, 2732, 2532
```

```
expr     Expression formed with + or - operators, symbols and hex numbers
         (symbols have the form "#symbol)
```

Figure 3-36. Verify Data in PROM Help Display

WINDOW MODE COMMAND

WIndow mode parameter

FUNCTION:

The Window Mode command specifies the type of display to be selected for viewing emulator memory. Only the first letter of the mode is entered.

OPERATION:

The Window mode command affects the current window. The current window is indicated by a reverse video line spanning the window at the current address. Each window may be set individually by redefining the active window with the TAB key.

Offset addressing, described in the help display below, does not apply to the 6809E Slave Emulator.

For further information regarding this command, please refer to the **SLAVE LOGIC ANALYZER MANUAL (2302-5012).**

WIndow mode

Window mode command operands:

Symbolic	Display memory data as disassembled symbolic instructions.
Hex	Display memory data as hexadecimal bytes and ASCII characters.
Cycle data	Display raw (minimally formatted) logic analyzer trace data.
Waveform	Display signals as waveforms from logic analyzer trace data.
Execution	Display execution trace from logic analyzer trace data.
Absolute	Use absolute addresses, referencing entire target address space.
Offset	Use 16-bit offset addresses, referencing a 64K segment beginning at the window's offset base address.

** Absolute/offset mode applies to memory displays, not to Analyzer traces.

Figure 3-37. Window Mode Help Display

WRITE COMMAND

WRite from

FUNCTION:

The WRite command writes the contents of memory to an object file.

OPERATION:

addr is the first address relative to the segment base address from which to begin writing. TAB defaults to 0. The default address appears at the tab position on the command line.

```
Prompts for:
                addr
write from
                <TAB>

                addr[,]
write to
               <TAB>
```

addr is the address expression relative to the segment base address at which to end writing. <TAB> defaults to X'FFFF'. The default address appears at the tab position on the command line.

"," continues Write from/to cycle if multiple memory areas are to be written to one file.

```
                addr
start addr=
               <TAB>
```

addr is the execution address expression of the file measured relative to the segment base address. <TAB> defaults to the start address specified in the last file loaded. The default address appears at the tab position on the command line.

```
write into file:    filename
```

If "[old file]" appears and is confirmed, the old file is overwritten. If "[new file]" appears and is confirmed, the file is created and written.

```
WRite from


Write
      address space     Address space name or <RETURN>
                        [requested only if target processor allows address
                        spaces]

      from              expr      Start address of block to write
                        TAB       Default start address:  0

      to                expr      End address of block to write
                        TAB       Default end address:  highest target address

      start addr =      expr      Execution start address
                        TAB       Default execution start addr from last file
                                  loaded

      into file         filename  Name of file to write

**Delimit end addr with "," to enter another block description
```

```
filename      Name of an object file
expr          Expression formed with + or - operators, symbols and hex numbers
              (symbols have the form "#symbol"
```

Figure 3-38. Write Command Help Display

SPECIAL KEYS

<BACKSPACE>	Deletes one character at a time from the command line.
<CAN>	Deletes the entry on the command line. Cancels the current command or subcommand.
<LOAD>	Returns control to the ADS terminal bootstrap. All boot level commands are allowed at this point.
<RESET>	Resets both the hardware and the software for the ADS and all Slave Emulators attached to it. Control of the display and keyboard is returned to the Emulator Executive.
<RETURN>	Multiple functions including entering a command; updating of the register, stack and memory displays to show the present status of the specified emulator; in some cases, terminating a command or a subcommand; unmapping a block of memory. Refer to specific commands for details.
<TAB>	Redefines the current display window when used as a command. The current window is indicated by a reverse video line across that window. All commands which affect memory begin at the address in the reverse video line of the current window. <TAB> also supplies a default value in many commands when used as an operand. <TAB> may be used in conjunction with the Find or STore commands, the PROM length subcommand and PROM type, PROM interleaving factor, Load command filename, Load offset, WRite block start addresses, WRite block end address, WRite execution start address, WRite filename, break parameters, etc. to invoke a default value. Refer to these commands individually for specific instructions.
<STEP>	Executes one user program instruction at a time. Execution begins at the address contained in the program counter. To allow the current window to track single step program execution, the window must be set to track the instruction pointer using the Display command.

<?>

Provides help displays to explain operation of Slave Emulator commands. When entered on a blank command line, a list of all available commands is displayed. Any subsequent keystroke returns to the previous display. Refer to **Figure 3-2.**

When <?> is entered following a command on the command line, explanatory information specific to that command is displayed. Completion of the command or subcommand replaces the help display with an appropriate display.

<;>

The semicolon allows the user to put comments in the command files. These are not executed, but merely echoed to the display.

F1

This is a special function key that appends the current screen image to the file specified in the last "Specify screen write options" command. It has no effect if the user has not completed this command since last jumping to the Slave Emulator software.

DISPLAY CONTROL KEYS

<↑>

The up arrow backs up the current location of the current window one instruction for a symbolic window or one line for a hex window.

<↓>

The down arrow advances the current location of the current window one instruction for a symbolic window or one line for a hex window.

<→>

The right arrow advances the current location of the current window one byte.

<←>

The left arrow backs up the current location of the current window one byte.

<+>

The plus character moves forward one page in memory. Refer to the Display command for definition of page size.

<->

The minus character moves back one page in memory.

GenRad

DSD SERVICE LOCATIONS

UNITED STATES AND CANADA

CO	300 Baker Avenue Concord, MA 01742 617/369-4400 or 617/646-7400 TWX: 710-347-1051	Conn., Maine, Mass., N.H., R.I., VT.
CHO	1083 East State Parkway Schaumburg, IL 60195 312/843-5580 TWX: 910-291-1209	IL., Iowa, Minn., Ohio, MO., N.D., S.D., KY., Ind., Mich., W.PA., Wis., Neb.
DEN	13132 St. Paul Drive Thornton, CO 80241 303/457-9147 (Ans. Serv.)	Colo., Mont., Wyoming, N.M., Utah
DSD	5730 Buckingham Parkway Culver City, CA 90230 (Factory) 213/641-7200; TWX: 910-328-7202	Factory Field Support
DTX	1121 Rockingham, Suite 100, Richardson, TX 75080 214/234-3357; TWX: 910-867-4771	Ark., Kan., LA., Tex., Okla.
FLO	3751 Maguire Blvd Suite 170, Orlando, FL 32803 305/894-4303; TWX: 810-850-0270	Ala., GA., Fla., N.C., S.C., Tenn., Miss.
LAO	17631 Armstrong Avenue P.O. Box 19500,Irvine, CA 92714 714/540-9830; TWX: 910-595-1762	S.CA., Ariz., S.Nev.
NYO	22-08 Route 208 Fair Lawn, NJ 07410 201/797-8001 (NJ); 212/964-2722 (NY) TWX: 710-988-2205	Del., E.PA., N.J., N.Y.
SFO	2855 Bowers Avenue, Santa Clara, CA 95051 408/727-4400; TWX: 910-338-0291	N.CA., Idaho, N. Nev., Wash., Ore.
WO	1701 Research Blvd, Rockville, MD 20850 301/424-6224; TWX: 710-828-9783	MD., VA., W.VA., D.C.
GRC	307 Evans Avenue Toronto, Ontario, Canada M8Z 1K2 416/252-3395; TELEX: 06-967624	All Canada

SERVICE LOCATIONS

EUROPE

DENMARK	Mr. Steen Schulstad BLT Agenturer A/S GL. Koge Landevej 55 DK-2500 Valby	Telephone: (01) 16 11 00 Telex: 855-16279 BITEK DK
SWEDEN	Mr. Stig Svensson, General Mgr. Mr. Gosta Olsson, Sales Mgr. Lagercrantz Electronik AB, Box 48 S-194 21 Upplands Vasby	Telephone: 760 861 20 Telex: 854-11275
FINLAND	Mr. Erik Hilden, Director Oy Emmett Ab Meterorinkatu 3 C-D 02210 ESPOO 21, Finland	Telephone: 009358/0882044 Telex: 857-22216
NORWAY	Mr. T.S. Fjeldstad, Director Bergman Instrumentering AS P.O. Box 129 Veitvet Sven Oftedals Vei 10 Oslo 5, Norway	Telephone: 16 22 10 Telex: 856-17271
NETHERLANDS	Mr. Michael Houdijk C.N. Rood BV Cort Van de Lindenstraat 11-13 2280 Rijskijk	Telephone: 070 996 360 Telex: 844-31238
FRANCE	Mr. Alberto Franzetti, AMM 6 Avenue Du General DeGaulle 78150 Le Chesnay	Telephone: 945 91 13 Telex: 842-698376
SPAIN	Mr. Francisco Herbada, General Mgr. Mr. Miguel Angel Ferrando, Sales Mgr. Hispano Electronica S.A. Poligono Industrial Urtinsa Apartado Correos 48 Alcorcon, Madrid	Telephone: 619 41 08 Telex: 831-22404
SWITZERLAND	Mr. Peter Multhanner, Sales Mgr. Mr. Ernst Schmid, General Mgr. Mr. Hans Wagenmakers GenRad (Schweiz) AG Drahzugstrasse 18, Postfach 8032 Zurich, Switzerland	Telephone: 55 24 20 Telex: 845-53638
AUSTRIA	Mr. Gunther Graf Kontron Elektronik A-2345 Brunn A. Geb Industriestrasse B13 Vienna, Austria	Telephone: 02236/866310 Telex: 847-79337

ITALY	Mr. Germano Fanelli Celdis Italiana Cinisello Balsamo 20092 Via Flll Gracchi, 36	Telephone: 612 00 41 Telex: 843-334883
GERMANY	Mr. Guido Negele, President Kontron Messtechnik GMBH Breslauer Strasse 2 D-8057 Eching Munich, W. Germany	Telephone: 089 3 1901 217 Telex: 841-522122
ENGLAND	Mr. David Findlay Kontron, Ltd. Campfield Road St. Albans AL1 5JG England	Telephone: Telex: