SECTION III

ASSEMBLY LANGUAGE

SECTION III

ASSEMBLY LANGUAGE

This section gives details on the Disk EDTASM assembly language. It does not explain the 6809 mnemonics, however, since there are many books available on the 6809.

To learn about 6809 mnemonics, read one of the books listed in "About This Manual." If you need more technical information on the 6809, read:

MC6809-MC6809E
8-Bit Microprocessor Programming Manual
Motorola, Inc.

Chapter 10/ Writing the Program

Chapter 3 gives a general description of assembly language instructions. This chapter describes them in detail.

The 6809 Registers

The 6809 contains nine temporary storage areas that you may use in your program:

REGISTER	SIZE	DESCRIPTION
A	1 byte	Accumulator
B	1 byte	Accumulator
D	2 bytes	Accumulator (a combination of A and B)
DP	1 byte	Direct Page
CC	1 byte	Condition Code
PC	2 bytes	Program Counter
X	2 bytes	Index
Y	2 bytes	Index
U	2 bytes	Stack Pointer
S	2 bytes	Stack Pointer

Table 2. 6809 Registers

Registers A and **B** can manipulate data and perform arithmetic calculations. They each hold one byte of data. If you like, you can address them as D, a single 2-byte register.

Register DP is for direct addressing. It stores the most significant byte of an address. This lets the processor directly access an address with the single, least significant byte.

Registers X and **Y** can each hold two bytes of data. They are mainly for indexed addressing.

Register PC stores the address of the next instruction to be executed.

Registers U and **S** each hold a 2-byte address that points to an entire "stack" of memory. This address is the top of the stack + 1. For example, if Register U contains 0155, the stack begins with Address 154 and continues downwards.

The processor automatically points Register S to a stack of memory during subroutine calls and interrupts. Register U is solely for your own use. You can access either stack with the PSH and PUL mnemonics or with indexed addressing.

Register CC is for testing conditions and setting interrupts. It consists of eight "flags." Many mnemonics "set" or "clear" one or more of these flags. Others test to see if a certain flag is set or clear.

This is the meaning of each flag, if set:

C (Carry), Bit 0 — an 8-bit arithmetic operation caused a carry or borrow from the most significant bit.

V (Overflow), Bit 1 — an arithmetic operation caused a signed overflow.

Z (Zero), Bit 2 — the result of the previous operation is zero.

N (Negative), Bit 3 — the result of the previous operation is a negative number.

I (Interrupt Request Mask), Bit 4 — any requests for interrupts are disabled.

H (Half Carry), Bit 5 — an 8-bit addition operation caused a carry from Bit 3.

F (Fast Interrupt Request Mask), Bit 6 — any requests for fast interrupts are disabled.

E (Entire Flag), Bit 7 — all the registers were stacked during the last interrupt stacking operation. (If not set, only Registers PC and CC were stacked.)

Assembly Language Fields

You may use four fields in an assembly language instruction: label, command, operand, comment. In this instruction:

```
START    LDA    #$F9         GETS CHAR
```

START is the label. LDA is the command. #$F9+1 is the operand. GETS CHAR is the comment.

The comment is solely for your convenience. The assembler ignores it.

The Label

You can use a symbol in the label field to define a memory address or data. The above instruction uses START to define its memory address.

Once the address is defined, you can use START as an operand in other instructions. For example:

```
    BNE        START
```

branches to the memory address defined by START.

The assembler stores all the symbols, with the addresses or data they define, in a "symbol table," rather than as part of the "executable program." The symbol can be up to six characters.

The Command

The command can be either a pseudo op or a mnemonic.

Pseudo ops are commands to the assembler. The assembler does not translate them into opcodes and does not store them with the executable program. For example:

```
    NAME        EQU        $43
```

defines the symbol NAME as $43. The assembler stores this in its symbol table.

```
                ORG        $3000
```

tells the assembler to begin the executable program at Address $3000.

```
    SYMBOL      FCB        $6
```

stores 6 in the current memory address and labels this address SYMBOL. The assembler stores this information in its symbol table.

Mnemonics are commands to the processor. The assembler translates them into opcodes and stores them with the executable program. For example:

```
    CLRA
```

tells the processor to clear Register A. The assembler assembles this into opcode number $4F and stores it with the executable program.

The next chapter shows how to use pseudo ops. *Reference L* lists the 6809 mnemonics.

The Operand

The operand is either a memory address or data. For example:

```
    LDD                #3000+COUNT
```

loads Register D with $3000 plus the value of COUNT. The operand, #$3000 + COUNT, specifies a data constant.

The assembler stores the operand with its opcode. Both are stored with the executable program.

Operators

The plus sign (+) in the above operand (#3000 + COUNT) is called an operator.

You can use any of the operators described in *Chapter 9*, "Using the ZBUG Calculator," as part of the operand.

Addressing Modes

The above example uses the # sign to tell the assembler and the processor that $3000 is data. When you omit the # sign, they interpret $3000 in a different "addressing mode."

Example:

```
    LDD                $3000
```

tells the assembler and processor that $3000 is an address. The processor loads D with the data contained in Address $3000 and $3001.

Each of the 6809 mnemonics lets you use one to six addressing modes. These addressing modes tell you:

- If the processor requires an operand to execute the opcode
- How the assembler and processor will interpret the operand

1. Inherent Addressing

There is no operand, since the instruction doesn't require one. For example:

```
SWI
```

interrupts software. No operand is required.

```
CLRA
```

clears Register A. Again, no operand is required. Register A is part of the instruction.

2. Immediate Addressing

The operand is *data*. You must use the # sign to specify this mode. For example:

```
          ADDA      #$30
```

adds the value $30 to the contents of Register A.

```
DATA      EQU       $8004
          LDX       #DATA
```

loads the value $8004 into Register X.

```
          CMPX      #$1234
```

compares the contents of Register X with the value 1234.

3. Extended Addressing

The operand is an *address*. This is the default mode of all operands.

(Exception: If the first byte of the operand is identical to the direct page, which is 00 on startup, it is directly addressed. This is an automatic function of the assembler and the processor. You need not be concerned with it if you're a beginner.)

For example:

```
          JSR       #$1234
```

jumps to Address $1234.

```
SPOT      EQU       $1234
          STA       SPOT
```

stores the contents of Register A in Address $1234.

If the instruction calls for data, the operand contains the address where the data is stored.

```
          LDA       $1234
```

does not load Register A with $1234. The processor loads A with whatever data is in Address $1234. If $06 is stored in Address $1234, Register A is loaded with $06.

```
          ADDA      $1234
```

adds whatever data is stored in Address $1234 to the contents of Register A.

```
          LDD       $1234
```

loads D, a 2-byte register, with the data stored in memory addresses $1234 and $1235.

You can use the > sign, which is the sign for extended addressing, to force this mode. (See "Direct Addressing.")

Extended Indirect Addressing.

The operand is the *address* of an *address*. This is a variation of the extended addressing mode. The [ ] signs specify it. (Use (SHIFT) (↓) to produce the [sign and (SHIFT) (→) to produce the] sign.)

In understanding this mode, think of a treasure hunt game. The first instruction is "Look in the clock." The clock contains the second instruction, "Look in the refrigerator."

Examples:

```
          JSR       [$1234]
```

jumps to the address contained in Addresses $1234 and $1235. If $1234 contains $06 and $1235 contains $11, the effective address is $0611. The program jumps to $0611.

```
SPOT      EQU       $1234
          STA       [SPOT]
```

stores the contents of Register A in the address contained in Addresses $1234 and $1235.

```
          LDD       [$1234]
```

loads D with the data stored in the address that is stored in Addresses $1234 and $1235.

This is a good mode of addressing to use when calling ROM routines. For example, the entry address of the POLCAT routine is contained in Address $A000. Therefore, you can call it with these instructions:

```
POLCAT    EQU       $A000
          JSR       [POLCAT]
```

If a new version of ROM puts the entry point in a different address, your program still works without changes.

4. Indexed Addressing

The operand is an *index register* which points to an

address. The *index register* can be any of the 2-byte registers, including PC. You can augment it with:

- A constant or register offset
- An auto-increment or auto-decrement of 1 or 2

The comma (,) indicates indexed addressing.

As an example, load X, a 2-byte register, with $1234:

```
        LDX     #$1234
```

You can now access Address $1234 through indexed addressing. This instruction:

```
        STA     ,X
```

stores the contents of A in Address $1234

```
        STA     3,X
```

stores the contents of A in Address $1237, which is $1234 + 3. (The number 3 is a constant offset.)

```
SYMBOL  EQU     $4
        STA     SYMBOL,X
```

stores the contents of A in Address $1238, which is $1234 + SYMBOL. (SYMBOL is a constant offset.)

```
        LDB     #$5
        STA     B,X
```

stores the contents of A in Address $1239 which is $1234 + the contents of B. (B is a register offset. You can use either of the accumulator registers as a register offset.)

```
        STA     ,X+
```

This instruction does two tasks: (1) stores A's contents in Address $1234 (the contents of X) and then (2) increments X's contents by one, so that X contains $1235.

```
        STA     ,X++
```

(1) stores A's contents in Address $1235 (the current contents of X) and then (2) increments X's contents by two to equal $1237.

```
        STA     ,--X
```

(1) decrements the current contents of X by two to equal $1235 ($1237 – 2) and then (2) stores A's contents in Address $1235.

As we said above, you can use PC as an index register. In this form of addressing, called program counter relative, the offset is interpreted differently. For example:

```
SYMBOL  FCB     0
        LDA     SYMBOL,PCR
```

While assembling the program, the assembler *subtracts* the contents of Register PC from the offset:

```
        LDA     SYMBOL-PC,PCR
```

While running the program, the processor *adds* the contents of Register PC to the offset. This causes A to be loaded with SYMBOL.

This seems to be the same as extended addressing. But, by using program counter relative adressing, you can relocate the program without having to reassemble it.

Indexed Indirect Addressing.

The operand is an *index register* which points to the *address* of an *address*. This is a variation of indexed addressing.

For example, assume that :

- Register X contains $1234
- Address $1234 contains $11
- Address $1235 contains $23
- Address $1123 contains $64

This instruction:

```
        LDA     [,X]
```

loads A with 64. (Register X points to the addresses of the address. This address is storing 6, the required data.)

```
        STA     [,X]
```

stores the contents of A in Address $1123. (Register X points to the addresses, $1234 and $1235, of the effective address, $1123.)

5. Relative Addressing

The assembler interprets the operand as a *relative address*. There is no sign to indicate this mode. The assembler automatically uses it for all branching instructions.

For example, if this instruction is located at Address $0580:

```
        BRA     $0585
```

The assembler converts $0585 to a relative branch of +3 (0585-0582).

This mode is invisible to you unless you get a BYTE OVERFLOW error, which we discuss below. Because the processor uses this mode, you can relocate your

program in memory without changing any of the branching instructions.

The BYTE OVERFLOW error means that the relative branch is outside the range of –128 to +127. You must use a long branching instruction instead. For example:

```
LBRA        $0600
```

allows a relative branching range of –32768 to +32767.

6. *Direct Addressing*

In this mode, the operand is *half of an address*. The other half of the address is in Register DP:

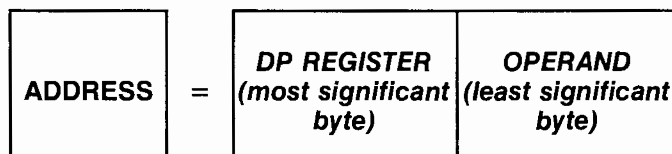

Figure 7. Direct Addressing

The assembler and the processor use this mode automatically whenever they approach an operand whose first byte is what they assume to be a "direct page" (the contents of Register DP). Until you change the direct page, the assembler and the processor assume it is 00.

For example, both of these instructions:

```
JSR         $0015
JSR         $15
```

cause a jump to Address $0015. In both cases, the assembler uses only 15 as the operand, not 00. When the processor executes them, it gets the 00 portion from Register DP and combines it with $15. (On startup, DP contains 0, as do all the other registers.)

Because of direct addressing, all operands beginning with 00, the direct page, consume less room in memory and run quicker. If most of your operands begin with $12, you might want to make $12 the direct page.

To do this, you first need to tell the assembler what you are doing, by putting a SETDP pseudo-operation in your program:

```
SETDP       $12
```

This tells the assembler to drop the $12 from all operands that begin with $12. That is, the assembler assembles the operand "1234" as simply "34".

Then, you must load Register DP with $12. Since you can use LD only with the accumulator registers, you have to load DP in a round-about manner:

```
LDB         #$12
TFR         B,DP
```

Now the direct page is $12, rather than 00. The processor executes all operands that begin with $12 (rather than 00) in an efficient, direct manner.

The assembler uses direct addressing on all operands whose first byte is the same as the direct page. You can denote direct addressing with the < sign if you want to document or be sure that direct addressing is being used.

For example, if the direct page is $12:

```
JSR         <$15
```

jumps to Address $1215. This instruction documents that the processor uses direct addressing.

Similarly, you might want to use the > sign to force extended addressing. For example:

```
JSR         >$1215
```

jumps to Address $1215. The assembler and processor use both bytes of the operand.

To learn more about 6809 addressing modes, read one of the books listed at the beginning of this manual.

Chapter 11/ Using Pseudo Ops

As discussed earlier, pseudo ops direct the assembler. You can use them to:

- Control where the program is assembled
- Define symbols
- Insert data into the program
- Change the assembly listing
- Do a "conditional" assembly
- Include another source file in your program

Pseudo ops are unique to the assembler you are using. Other 6809 assemblers may not recognize the Disk EDTASM pseudo ops.

The Disk EDTASM pseudo ops make it easier for you to program. This chapter shows how to use pseudo ops.

Controlling Where the Program is Assembled

The Disk EDTASM has two pseudo ops that control where the program is assembled:

- ORG, sets the first location
- END, ends the assembly

ORG

ORG *expression*

Tells the assembler to begin assembling the program at *expression*. Example:

```
        ORG                     $1800
```

tells the assembler to start assembling the program at Address $1800.

You can put more than one ORG command in a program. When the assembler arrives at the new ORG, it begins assembling at the new *expression*.

END

END *expression*

Tells the assembler to quit assembling the program. The *expression* option lets you store the program's start address. Use END as the last instruction in all your assembly language programs.

Example:

```
                ORG     $1800
DATA            FCC     'This is some data'
START           LDA     DATA
                 .
                 .
                END     START
```

The END pseudo op quits the assembly and stores the program's entry address (the value of START) on disk. When you load the program, the processor knows to start executing at START (the LDA instruction) rather than at DATA (the FCC instruction).

FCC is a pseudo op explained later in this chapter.

Defining Symbols

Symbols make it easy to write a program and also make the program easy to read and revise. The Disk EDTASM has two pseudo ops for defining symbols:

- EQU, for defining a constant value
- SET, for defining a variable value

EQU

symbol* EQU *expression

Equates *symbol* to *expression*. Examples:

```
CHAR        EQU         $F9
```

equates CHAR to $F9.

```
SCREEN      EQU         $5ØØ
            LDX         #SCREEN
```

equates SCREEN to $500. The next instruction loads X with $500.

EQU helps set the values of constants. You can use it anywhere in your program.

SET

symbol* SET *expression

Sets *symbol* equal to *expression*. You can use SET to reset the symbol elsewhere in the program. Example:

```
SYMBOL      SET         25
```

sets SYMBOL equal to 25. Later in the program, you can reset SYMBOL.

```
SYMBOL      SET         SYMBOL+COUNT
```

now SYMBOL equals 25 + COUNT.

Inserting Data into Your Program

The Disk EDTASM has four pseudo ops that make it simple for you to reserve memory and insert data in your program:

- RMB, for reserving areas of memory for data
- FCB, for inserting one byte of data in memory
- FDB, for inserting two bytes of data in memory
- FCC, for inserting a string of data in memory

Remember that the processor cannot "execute" a block of data in your program. If you use these pseudo ops:

- Use them at the end of your program (just before the END instruction), or
- Precede them with an instruction that jumps or branches to the next "executable" instruction.

RMB

symbol* RMB *expression

Reserves *expression* bytes of memory for data. Example:

```
BUFFER      RMB         256
```

reserves 256 bytes for data, starting at Address BUFFER.

```
DATA        RMB         6+SYMBOL
```

reserves 6 + SYMBOL bytes for data beginning at Address DATA.

FCB

symbol* FCB *expression

Stores a 1-byte *expression* in memory at the current address. The *symbol* is optional.

Examples:

```
DATA        FCB         $33
```

stores $33 in Address DATA.

```
FACTOR      FCB         NUM/2
            LDA         FACTOR
```

stores NUM/2 in Address FACTOR, then, loads NUM/2 into Register A.

FDB

symbol* FDB *expression

Stores a 2-byte *expression* in memory starting at the current address. The symbol is optional. Example:

```
DATA        FDB         $3322
```

stores $3322 in Address DATA and DATA + 1.

FCC

symbol* FCC *delimiter string delimiter

Stores an ASCII string in memory, beginning at the current address. The *symbol* is optional. The *delimiter* can be any character.

Examples:

```
TABLE   FCC     /THIS IS A STRING/
```

stores the ASCII codes for THIS IS A STRING in memory locations, beginning with TABLE.

```
NAME     FCC     'Dylan'
         FCB     $0D
         LDB     #NAME
INIT     LDA     NAME
         .
         .
         INCB
         CMPA    NAME
         BNE     INIT
```

The first instruction stores "Dylan" in the five memory addresses beginning with NAME. The next instructions process this data.

Changing the Assembly Listing

You can use three pseudo ops to change the listing the assembler prints for you:

- TITLE, inserts a title at the top of each listing page
- PAGE, ejects the listing to the next page
- OPT, turns on or off the switches that determine how the assembler lists "macros" (Macros are discussed in the next chapter.)

TITLE *string*

Tells the assembler to print the first 32 characters of the *string* at the top of each assembly listing page. Example:

```
TITLE    Budget Program
```

causes the assembler to print Budget Program as the title of each page in the assembly listing.

PAGE

Starts a new page if the assembly listing is being printed on the line printer. Example:

```
PAGE
```

tells the assembler to eject the listing to the next page.

OPT

OPT *switch, switch, . . .*

Causes the assembler to use the specified *switches* when printing its listing. You can specify these *switches* with OPT:

MC	List macro calls (default)
NOMC	Do not list macro calls
MD	List macro definitions (default)
NOMD	Do not list macro definitions
MEX	List macro expansions
NOMEX	Do not list macro expansions (default)
L	Turn on the listing (default)
NOL	Turn off the listing

Example:

```
OPT      MEX
```

Causes the assembler to list the macro expansions in its listing. (Macros are discussed in the next chapter.)

Conditional Assembly

You may want to execute a certain section of your program only if a certain condition is true. The Disk EDTASM lets you set up a "conditional" section of your program, using these two pseudo ops:

COND

COND *condition expression*

Assembles the following instructions only if the *expression* is true (non-zero). If not true (zero), the assembler goes to the instruction that immediately follows the ENDC instruction.

Only these operators are recognized in a condition expression: +,–,/,*. See ENDC below for an example.

ENDC

ENDC

Ends a conditional assembly, initiated by COND.

Examples:

```
COND SYMBOL
 .
 .
ENDC
```

assembles the lines between COND SYMBOL and ENDC only if SYMBOL is not equal to zero.

```
COND VALUE2-VALUE1
 .
 .
ENDC
```

assembles the lines between VALUE2-VALUE1 only if VALUE2-VALUE1 are not equal (which causes the result to be a non-zero value).

Including Other Source Files

To let you load another source file and include it in your program, the Disk EDTASM offers an INCLUDE pseudo op.

INCLUDE

INCLUDE *filespec*

Inserts *filespec*, a file of source assembly language instructions, at the point where INCLUDE appears in the program. The assembler assembles the entire included file before assembling the next instruction.

Example:

```
INCLUDE          ROUTINE/SRC
```

inserts and assembles ROUTINE/SRC, a source file, before assembling the next instruction.

```
INCLUDE          SUB1/SRC
INCLUDE          SUB2/SRC
```

inserts and assembles SUB1, then inserts and assembles SUB2, then proceeds with the next instruction.

Chapter 12/ Using Macros

A macro is like a subroutine. It lets you call an entire group of instructions with a single program line. This helps when you want to use the same group of instructions many times in the program.

This chapter first tells how to use a macro. It then gives guidelines on the format of a macro.

How to Use a Macro

To use a macro, you must first define it. For example, you could define the entire sample program (from *Chapter 2*) as a macro named GRAPH.

After defining the macro, you can use its name the same way you use a mnemonic. Whenever the assembler encounters the macro's name, it expands it into the defined instructions.

Defining a Macro

To define a macro, you need to:

- Use MACRO (a pseudo op) to begin the macro definition and assign it a name.
- Use source instructions to define the macro.
- Use ENDM (a pseudo op) to end the macro definition.

This is an example of the sample program converted into a macro definition:

```
00030 GRAPH    MACRO
00100          LDA      #$F9
00110          LDX      #$400
00120 \.A      STA      ,X+
00130          CMPX     #$600
00140          BNE      \.A
00150 \.B      JSR      [$A000]
00160          BEQ      \.B
00180          ENDM
```

Line 30 names the macro as GRAPH, lines 50-160 define the macro, and line 180 ends the macro definition.

Notice the names of the symbols within the macro definition: \.A and \.B. If you do not use this format for naming symbols, you'll get a MULTIPLY DEFINED SYMBOL error when you call the macro more than once. (More on this later.)

Insert the above program using (SHIFT) (CLEAR) to generate the backslash character (\). Save the program on disk as MACRO1 and then delete it.

```
WD MACRO1 (ENTER)
D#:* (ENTER)
```

Calling a Macro

To call a macro, simply use the macro name as if it were a mnemonic. For example, this sample program calls GRAPH and then ends:

```
00110          ORG      $1200
00120 BEGIN    JMP      START
00130          FDB      DONE-BEGIN
00140 START    *
00150          INCLUDE  MACRO1/ASM
00160          GRAPH
00170          CLR      $71
00180          JMP      [$FFFE]
00190 DONE     *
00200          END
```

Line 150 loads MACRO1, the file containing the definition of GRAPH, and includes it in the source program. Line 160 calls the GRAPH macro.

To see how the assembler expands the GRAPHIC macro, insert this line:

```
00135          OPT      MEX
```

and assemble the program. The assembler listing shows how the assembler expands GRAPH into its defined instructions.

Note that the assembler has replaced \.A with A0000 and \.B with B0000. The zeroes indicate that this is the first expansion of the symbols in GRAPH. (In this case, this is the only expansion.)

Passing Values to a Macro

A convenient way to use a macro is to pass values to it. You can use a macro many times in your program, passing different values to it each time.

This is a definition of the GRAPH macro, slightly modified so that you can pass two values to it. Insert this program, save it as MACRO2 and then delete it.

```
00030  GRAPH2  MACRO
00100          LDA      \0
00110          LDX      \1
00120  \.A     STA      ,X+
00130          CMPX     #$600
00140          BNE      \.A
00150  \.B     JSR      [$A000]
00160          BEQ      \.B
00190          ENDM
```

The \0 and \1 are dummy values. The assembler replaces these numbers with the values you specify when you call GRAPH.

The following program calls GRAPH2 three times. Each time it passes two different sets of values:

```
00100          ORG      $1200
00110  BEGIN   JMP      START
00120          FDB      DONE-BEGIN
00130  START   *
00140          OPT      MEX
00150          INCLUDE  MACRO2/ASM
00160          GRAPH2   #$F9,#$400
00170          GRAPH2   #$F8,#$450
00180          GRAPH2   #$F7,#$500
00190          CLR      $71
00200          JMP      [$FFFE]
00210  DONE    *
00220          END
```

When the assembler expands the macro, it replaces the dummy values with the values passed by the macro call. For example, the second time GRAPH2 is called, the assembler replaces \0 with #$F8 and replaces \1 with #$450.

Assemble the above program. Note that each time the assembler expands GRAPH2, it replaces the \.A and \.B symbols with different symbol names: First A0000 and B0000, then A0001 and B0001, and finally A0002 and B0002.

If the assembler used the same symbol names in each expansion, it would be forced to assign different value to the symbols in each expansion. You would get a MULTIPLY DEFINED SYMBOL error.

Also, note the assembler has inserted an additional symbol, NARG, in the symbol table. NARG is always set to the number of values passed in the most recent macro call.

In the sample program, the symbol table shows that NARG is set to "2" at the end of the assembly. This shows that there were two values passed to GRAPH2 the last time it was called.

You might want to use NARG as a variable in your program. For example, you could conditionally assemble parts of a macro definition based on the current value of NARG.

To see the program run, assemble it to disk, press a key three times to see different graphics and then end the program.

Format of Macros

The remainder of this chapter gives details on the format to use in a macro definition and macro call.

Macro Definition

Beginning the Definition

Use this format for beginning the macro definition and assigning it a name:

symbol MACRO

symbol is the name of the macro. It is, of course, required.

Using Symbols in the Definition

Use this format to name any symbols you use within a macro definition:

\.*c*

c is an alpha character (A-Z). When the assembler expands the macro, it replaces \.*c* with:

cnnnn

nnnn is a 4-digit hexadecimal number that the assembler increments each time the assembler expands the macro.

For example, if you use the symbol \.M in the macro definition and you call the macro 10 times, the assembler replaces \.M with these symbol names:

```
1st expansion       M0001
2nd expansion       M0002
      .
10th expansion      M000A
```

You must use this symbol-name format when calling a macro more than once. Otherwise, you get MULTIPLY DEFINED SYMBOL errors.

Using Dummy Values in the Definition

Use this format for specifying dummy values within a macro definition:

n

n is an alphanumeric character (0-9,A-Z). The assembler replaces this dummy value with a corresponding value in the macro call line:

\0 is replaced with the 1st value
\1 is replaced with the 2nd value
.
.
\9 is replaced with the 10th value
\A is replaced with the 11th value
.
.
\Z is replaced with the 36th value

For example, this line in a macro definition:

```
LDA    \B
```

specifies \B as a dummy value. The assembler replaces \B with the 12th value in the macro call line. If the macro call line is:

```
ADD       NUM0,NUM1,NUM2,NUM3,NUM4,
NUM5,NUM6,NUM7,NUM8,NUM9,NUMA,NUMB
```

the assembler replaces \B with NUMB.

You do not need to assign macro call values to dummy values in consecutive order. For example:

```
          GRAPHX    #$F9,#$400,#$600
GRAPHX    MACRO
          LDX       \1
          LDY       \2
          LDA       \0
          LDB       \0
          ENDM
```

Here, the assembler replaces dummy value \1 with #$400, replaces dummy value \2 with #$600, and, in two lines, replaces dummy value \0 with #$F9. Note that you can pass a value to a macro more than once, as this example does with #$F9.

If there are more dummy values than values in a macro call, a byte overflow error results.

If there are more values than dummy values in a macro call, the extra values are ignored.

Be sure not to enclose dummy values in quotes. If you do this, the assembler treats them as ordinary characters.

Ending the Macro Definition

Use this format for ending the macro definition:

```
ENDM
```

You may not use a symbol to label this line. If you do so, you get a MISSING END STATEMENT error at the end of the assembly listing.

Macro Call

Use this format when passing values to a macro in a macro call line:

macro call   string1, string2, ...

macro call is the name of the macro.

string(s) is the value being passed to the macro. It can be 1 to 16 characters (any extra characters are ignored).

Each string, except the last, must be separated by a comma. The last string must be terminated by a comma, space, carriage return, or tab.

Each string may contain any characters except a carriage return. If a string contains a comma, space, tab, or left parenthesis, you must enclose it in parentheses. For example, in this macro call:

```
PRINT      (ABC,DEF)
```

the assembler interprets ABC,DEF as a single string. However, in this call:

```
PRINT      ABC,DEF
```

the assembler interprets ABC as one string and DEF as another.

Hints on Macros

- Remember to define a macro before calling it. If you call a macro without defining it, you get a BAD OPCODE error.

- We recommend storing all macro definitions in a file and then using INCLUDE to insert them into your main program.
- Do not use a mnemonic or pseudo op as a macro name. This causes the assembler to redefine the mnemonic or pseudo op according to the macro definition.
- If the macro definition has an error, you will not discover the error until you call the macro. The assembler waits until you call the macro before it assembles it.
- You cannot "nest" macro definitions. That is, one macro definition cannot call another.
- Using the same macro more than once uses a large amount of memory. Expand a large macro only once. When you want to use it again, call it as a subroutine.