Twenty Dollars

Cromemco®

68000 Macro Assembler

Instruction Manual

CROMEMCO, Inc.
280 Bernardo Avenue
Mountain View, CA. 94043

Part No. 023-4053

February 1983

Cromemco®

68000 Macro Assembler

Instruction Manual

CROMEMCO, Inc.
280 Bernardo Avenue
Mountain View, CA. 94043

Part No. 023-4053

February 1983

This manual was produced using a Cromemco System Three computer with a Cromemco HDD-22 Hard Disk Storage System running under the Cromemco Cromix® Operating System. The text was edited with the Cromemco Cromix Screen Editor. The edited text was proofread by the Cromemco SpellMaster™ Program and formatted by the Cromemco Word Processing System Formatter II. Camera-ready copy was printed on a Cromemco 3355B printer.

TABLE OF CONTENTS

LIST OF TABLES

Chapter 1

THE 68000 MACRO ASSEMBLER

The 68000 Macro Assembler translates 68000 assembly language mnemonics into object code that can be linked and executed under the Cromix Operating System. This Assembler gives you a great deal of flexibility in writing source code.

-- You can write macros to handle repetitive functions. These macro definitions can appear either in a single program or in a library where any number of programs can use them. See Macros in Chapter 3 for more information.

-- You can write general routines that include blocks of code that are needed only in certain cases. Then when you assemble the routines, you control what blocks of code are included and excluded through the use of IF statements. See Conditional Assembly in Chapter 3 for more information.

-- You can use either relocatable or absolute addressing. See Program Segments in Chapter 4.

-- You can write a program in as many modules as you wish and, as long as you assemble them with relocatable addresses, link them together. See Chapter 4.

This manual describes the features offered by the Assembler and how to use them. It is assumed that you already know the 68000 instruction set. If you don't, see your Cromemco dealer for books describing this instruction set.

The Assembler and the assembled programs can execute only on Cromemco DPU (Dual Processing Unit) based systems.

PREPARING A PROGRAM FOR EXECUTION

The preparation of an assembly language program is a two-step process. In the first, you prepare the source code for the Assembler to translate into object code. In the second, you prepare the object code for execution. This manual primarily deals with the first set of tasks, preparing the source code. In order to do this effectively, though, you must be aware of the steps and programs used in preparing both the source code and object code.

The description of these tasks that follows is meant to familiarize you with the steps and programs. Before you actually begin preparing a program, you should read the introductions of the manuals describing the programs that you plan to use.

As the first step, use either the Screen Editor or some other editor to enter the source code into a disk file. Then use the Assembler to produce an object code version of the source code. In the last step, use the Linker to link the module and produce executable code from the object code.

The Assembler Input and Output Files

The basic input for the Assembler is a disk file containing the assembly source code. This file contains the 68000 assembly operation codes and the pseudo operation codes that direct the Assembler. From this input, the Assembler typically produces a disk file containing the object code and a disk file containing the assembly listing. Error messages go to the console. (Error messages relating to problems with the source code are also part of the assembly listing.)

Inserting Code in a Module

To understand the major options for preparing your programs, you must understand the concept of a **module.** A module is the executable code ultimately produced from the source code in a single file. In the simplest case, a module is the complete program produced from the assembly of a single source code file.

In a more complex variation, you can have either the Assembler or Linker insert code from other files into the module. The inserted code becomes as much a part of the module as the original code; the Assembler and Linker treat the original and inserted code the same.

Inserting code has many of the same advantages as using macros and subroutines. You can, for example, write a subroutine, place it in a file, and then have it included in any module that needs it. You can do the same with data tables, macro definitions, or any other type of code you expect to use in more than one module.

You also might want to include code so it is in separate files, where it is easier to maintain and does not clutter up the primary source code file.

The Assembler provides several ways to include code:

- The *INCLUDE pseudo operation code inserts the entire contents of one file into another. The file can contain any type of code. You can nest included code up to eight levels deep. That is, a module can include code that, in turn, includes code and so forth up to eight include statements.

  A module can have any number of included blocks of code so long as no set of included code is the result of more than eight levels of included code.

- The *MACLIB pseudo operation code inserts all the macro definitions from a given file into the opcode table. Only the names of such macros are inserted, together with a pointer to their definition on the disk. When such a macro is called, the definition is stored in the memory the same way as if the macro were defined in source text. If a macro from a macro library is not called, it is not loaded.

- The *RELOBJ pseudo operation code inserts the entire contents of one object code file into the module as it is linked. The file can contain any type of object code.

  RELOBJ works the same way INCLUDE does, except that it inserts object code instead of source code into the module.

- The *RELLIB pseudo operation code differs from the other include pseudo operation codes because it does not insert a file of code. Rather, it names a file containing object code routines that the Linker is to search for definitions of unresolved global labels. The Linker inserts only those routines that define the global labels. The rest are not inserted. You may create an object library by just concatenating individual object files, or you may use the MAKLIB program to do it. The MAKLIB program will sort the modules in the correct

order (duplicating them if necessary), so that the linker will always find all required modules in a single search pass.

You can name any number of libraries with this pseudo operation code.

Chapter 7 describes the syntax of the pseudo operation codes just summarized.

A module does not have to contain all the code of a program. One of the Linker's major purposes is to link different modules together into a complete program. Chapter 4 describes the preparations you must take in writing a module to have it linked with other modules. The linker manual describes the Linker's ability to link modules together.

Correcting Errors in the Program

If there are errors in your code, you will find them after assembling a module, linking the program, or executing the program. When you do find errors, you can use the Debug program to execute the program, instruction by instruction, and substitute new instructions for the ones in error. Using Debug is optional; you can change the source code instead and should do so anyway once you have used Debug to find out what corrections should be made.

SYSTEM CALLS

A system call is an instruction in the source code that requests the operating system to perform a function. The system calls principally perform input and output to the disk drives, the terminals, the printers, and other peripherals in a system. Some system calls perform specialized functions such as requesting the date from the operating system.

All input and output should be done with system calls so that the programs can be independent of the requirements and arrangement of the input/output devices. This allows a program written for one Cromemco system to work on other Cromemco systems. It also allows the device drivers to be rewritten if necessary without requiring changes in the programs because the system calls to the device drivers do not need to change. If system calls were not used, device drivers would have to be written for every program.

To use a 68000 Cromix system call, load any needed parameters into registers and then make the call with the Jsys operation code. (Jsys is a special operation code included in the Cromemco 68000 Macro Assembler.) The Jsys operand specifies the function to be performed.

Two files supplied with the 68000 Macro Assembler, **jsysequ.asm** and **modeequ.asm**, provide equates for the system function names and mode options so that you do not have to memorize the numbers that the system actually uses. To use these files, include them in your program with the *INCLUDE pseudo operation code. Refer to Chapter 7.

This example shows the instructions needed to make a system call to ring the bell on a terminal:

```
*INCLUDE 'jsysequ.asm'
         :
         :
         :
        MOVE    #07H,D0      ; Load the ASCII value for
                             ;    terminal bell, 7,
                             ;    into register D0
        MOVE    #STDOUT,D1   ; Load the equated value
                             ;    for standard output
                             ;    channel
                             ;    into register D1
        JSYS    #_WRBYTE     ; Call Cromix to
                             ;    write a byte of data
```

The system calls preserve the contents of all registers except those containing return parameters.

Chapter 2

STATEMENT SYNTAX

The Cromemco 68000 Macro Assembler requires that all instructions and pseudo operation codes be written in the standard Assembler format shown here:

```
[label:] operation [operands] [; comments]
```

This chapter gives the basic limits on the use of the format. For the rules applying to particular instructions, refer to a 68000 user's manual. For the rules applying to particular pseudo operation codes, refer to Chapter 7.

The maximum line length accepted by the Assembler is 489 characters. (The last character must be a line feed marking the end of the line.)

BASIC SYNTAX

The following rules apply to all statements.

Field Delimiters

Each field in a statement must be separated from the other fields by one of these delimiters:

: The colon marks the end of the label if one appears in the statement. It must be used if the label does not begin in column 1 of the statement and is optional if the label does begin in column 1. You must use one of the following characters to mark the end of a label that begins in column 1: TAB (CONTROL-I), SPACE, semicolon (;), or RETURN.

Blank A blank space marks the end of the operation field and can be used to mark the end of a label beginning in column 1. You can insert as many spaces as you want between the fields of a statement so long as the statement is less than 489 characters long.

TAB A TAB character (CONTROL-I) can mark the end of the operation field and can be used to mark the end of a label beginning in column 1. You can insert as many TABs as you want between the fields of a statement so long as the statement is less than 489 characters long.

; A semicolon marks the beginning of a statement's comments. If a statement doesn't have comments, then the semicolon doesn't have to be used. The comment field extends from the semicolon to the end of the statement. If the semicolon is the first non-blank or non-TAB character on a line, the entire line is treated as a comment. (The REM pseudo operation can also be used to write a comment as described in Chapter 7.)

LF A line-feed character marks the end of a statement.

RETURN can precede a line-feed character, so that each line must be terminated by a LF or RETURN-LF sequence. Note that the Screen editor gives you a RETURN-LF pair when you press the RETURN key, but if you are going to produce the source text with a program, you are allowed to terminate lines by the LF character only.

Within these rules, the formatting of a statement is free form, and delimiters can be mixed at will to make the statement as readable as possible.

```
;    Program hello
;
*include        '/EQU/JSYSEQU.ASM'
;
START:  MOVE    #STDOUT,D1      ; write on standard output
        LEA     MSG,A0          ; point to message line
        JSYS    #_WRLINE        ; write it
        JSYS    #_EXIT          ; exit to operating system
;
MSG:    DC.B    'Hello, world\n\0'
        END     START
```

The colons following the labels in the example are superfluous because the labels start in column 1 and are included only to increase readability.

Acceptable Characters

The Assembler accepts any printable ASCII-encoded character in a statement. (What the Assembler accepts, however, and what works when the object code is executed can be two separate items.) More specifically, the Assembler accepts any hexadecimal value between 20 and 7E. It also accepts the CONTROL-I character (the TAB character), the RETURN character, and the line-feed character.

The Assembler doesn't distinguish between the upper and lower cases of letters. To the Assembler, these spellings are identical and produce the same object code:

```
ADD   Add   add   aDd
```

Even though the Assembler treats upper- and lower-case letters identically, it lists variation in case (e.g., ADD versus add) separately in the operation code and label cross-reference tables. If you are consistent in your use of upper and lower case, the use of labels and mnemonics in different places is readily apparent in the cross-references. For example, the labels in the source code could all be written in lower case, while the labels in macro definitions could all be written in upper case.

Backslash Characters for ASCII Strings

The Assembler interprets several backslash (\)-character combinations as control characters for printing ASCII strings. These combinations are:

COMBINATION	INTERPRETED AS	ASCII VALUE
\N	New line	0A
\L	Line feed	0A
\F	Formfeed	0C
\R	RETURN	0D
\B	Backspace	08
\T	TAB	09
\0	Null	00
\Xhh	Hex Digits	hex value

The Assembler interprets these combinations as ASCII control characters only when they are found within single quotes ('). The following example shows a message terminated by the combinations for a RETURN and a line feed:

```
ERRMSG:         DC.B    'INVALID ENTRY\R\N'
```

THE LABEL FIELD

When used, the label field contains a name to be equated with a value, usually the current value of the program counter. With a few pseudo operations, such as EQU, the value is an expression given in the operand field of the statement. The Assembler assigns the value of the program counter to any label appearing on a line that does not have an operation field. This statement:

```
LABEL1:
```

is equivalent to this one:

```
LABEL1          EQU     $
```

As described in Current Program Counter ($) in this chapter, the Assembler substitutes the value of the current program counter for the dollar sign when it is used as an operand.

The allowed use of labels varies from instruction to instruction and from pseudo operation code to pseudo operation code.

Labels may be as long as one line, but are truncated by the Assembler to the first 32 characters.

You can use the following characters in a label:

```
A - Z
a - z
0 - 9
$  (dollar sign)
_  (underscore)
?  (question mark)
@  (at sign)
```

The label field must be terminated by a colon, a SPACE, a TAB, or a RETURN. If the label field does not begin in column 1, it must be terminated by a colon.

The following examples show labels in statements (the labels appear in bold letters):

```
START:    LEA     COUNT,A2

USER_TABLE_NAME_FIELD    DS   15

LOAD:MOVE  2(A2,D3),D3   ; Load account no.

loop:     addq    #1,(A3)+

CONSTANT EQU      0FFA3H

SHIFT:    MACRO   \P1,\P2       ;; SHIFT BITS MACRO
```

The following labels are illegal because the Assembler considers them to be register names:

```
A0  A1  A2  A3  A4  A5  A6  A7
D0  D1  D2  D3  D4  D5  D6  D7
SP  SR  CCR
```

These labels are also illegal if written in lower case.

THE OPERATION FIELD

When it is used, the operation field contains either a 68000 instruction mnemonic or a pseudo operation code instruction. The Assembler recognizes all the standard 68000 instruction mnemonics.

Most operation codes may begin on any column of a line except column 1. The four pseudo operation codes that begin with an asterisk -- *INCLUDE, *MACLIB, *RELLIB, and *RELOBJ -- must begin in column 1. (See Chapter 7 for a description of the syntax of these pseudo operation codes.) If a label appears in the statement, a colon, SPACE or TAB must separate the label and the operation. A SPACE, TAB, semicolon, or RETURN ends the operation.

The following examples show instructions in statements (the instructions are printed in bold letters):

```
          AND.B     #11100100B,(A1)

NAME:     DS        23          ; Name field

          END

SHIFT:    MACRO     \P1,\P2     ;; SHIFT BITS MACRO

          LSR.L     #4,REGIS
```

The 68000 instruction set provides multiple forms of several instruction types. The ADD function, for example, comes as five separate instructions: the normal Add (ADD), the Add Address (ADDA), the Add Quick (ADDQ), the Add Immediate (ADDI), and the Add with Extend (ADDX). Because it can be troublesome to remember the different forms of these instructions, the Assembler allows you to specify the normal form of these instructions with the operands that are normally reserved for the more specialized forms.

Say, for example, you want to add 10 to the value specified by register A2. This operation requires that you use the Add Immediate, ADDI, instruction. You could just use the ADDI instruction:

```
ADDI      #10,(A2)
```

or you could use the normal ADD instruction:

```
ADD       #10,(A2)
```

When the Assembler encounters this instruction, it examines the operands and finds that they require an ADDI instruction, which the Assembler produces and places into the object file:

```
ADDI      #10,(A2)
```

The Assembler cannot substitute one specialized form for another form. If, for instance, you used this instruction:

```
ADDI      D2,(A2)
```

the Assembler would not substitute the ADD instruction for the incorrectly used ADDI instruction. Instead, it would flag the statement as an error.

Table 2-1 lists the instructions that the Assembler examines and the more specialized forms that it can substitute. Table 2-2 lists the forms of these instructions that the Assembler cannot substitute; that is, you must use this form in the source code. For example, the Assembler cannot recognize an Add Extended, ADDX, instruction from the operands. You must specify ADDX when you want to do an extended add.

Table 2-1: ALTERNATE FORMS OF OPERATION CODES THE ASSEMBLER CAN AUTOMATICALLY SUBSTITUTE

Standard Form	Variation	Description
ADD	ADD	Add
	ADDA	Add address
	ADDQ	Add quick
	ADDI	Add immediate
AND	AND	Logical AND
	ANDI	AND immediate
CMP	CMP	Compare
	CMPA	Compare address
	CMPM	Compare memory
	CMPI	Compare immediate
EOR	EOR	Exclusive OR
	EORI	Exclusive OR immediate
MOVE	MOVE	Move
	MOVEA	Move address
	MOVEQ	Move quick
OR	OR	Logical OR
	ORI	OR immediate
SUB	SUB	Subtract
	SUBA	Subtract address
	SUBI	Subtract immediate
	SUBQ	Subtract quick

Table 2-2: ALTERNATE FORMS OF OPERATION CODES THE ASSEMBLER CANNOT AUTOMATICALLY SUBSTITUTE

Standard Form	Variation	Description
ADD	ADDX	Add with extend
MOVE	MOVE from SR	Move from status register
	MOVE to SR	Move to status register
	MOVE to CCR	Move to condition codes
	MOVE to USP	Move to user stack pointer
NEG	NEGX	Negate with extend
SUB	SUBX	Subtract with extend

The following sections provide more information on how the Assembler substitutes instructions.

Automatic Substitution of ADD Instructions

The Assembler can substitute specialized versions of the ADD instruction as follows:

IF YOU SPECIFY:	THE ASSEMBLER PRODUCES:
Normal ADD	
ADD <ea>,Dn	ADD <ea>,Dn
ADD D1,D2	ADD D1,D2
ADD Dn,<ea>	ADD Dn,<ea>
ADD D3,(A5)	ADD D3,(A5)
ADD Address	
ADD <ea>,An	ADDA <ea>,An
ADD COUNT,A1	ADDA COUNT,A1
ADD Immediate	
ADD #data,<ea>	ADDI #data,<ea>
ADD #50,D2	ADDI #50,D2
ADD Quick	
ADD #data,<ea>	ADDQ #data,<ea>
ADD #3,D4	ADDQ #3,D4

Because the Assembler cannot automatically substitute the ADDX instruction for the ADD instruction, you must explicitly use the ADDX form when you want to perform an add with extend.

Automatic Substitution of AND Instructions

The Assembler can substitute the AND immediate form of the AND instruction as follows:

IF YOU SPECIFY:	THE ASSEMBLER PRODUCES:
Normal AND	
AND <ea>,Dn	AND <ea>,DN
AND D4,D5	AND D4,D5
AND Dn,<ea>	AND Dn,<ea>
AND D3,(A3)	AND D3,(A3)
AND Immediate	
AND #data,<ea>	ANDI #data,<ea>
AND.B #100B,(A1)	ANDI.B #100B,(A1)

Automatic Substitution of Compare Instructions

The Assembler can substitute specialized versions of the compare instruction as follows:

IF YOU SPECIFY:	THE ASSEMBLER PRODUCES:
Normal Compare	
CMP <ea>,DN	CMP <ea>,DN
CMP D1,D4	CMP D1,D4
Compare Address	
CMP <ea>,An	CMPA <ea>,An
CMP A1,A2	CMPA A1,A2
Compare Immediate	
CMP #data,<ea>	CMPI #data,<ea>
CMP #45FAh,D3	CMPI #45FAh,D3
Compare Memory	
CMP (Ay)+,(Ax)+	CMPM (Ay)+,(Ax)+
CMP (A1)+,(A2)+	CMPM (A1)+,(A2)+

Automatic Substitution of Exclusive OR Instructions

The Assembler can substitute specialized versions of the exclusive OR instruction as follows:

IF YOU SPECIFY:	THE ASSEMBLER PRODUCES:
Normal Exclusive OR	
EOR Dn,<ea>	EOR Dn,<ea>
EOR D3,(A4)+	EOR D3,(A4)+
Exclusive OR Immediate	
EOR #data,<ea>	EORI #data,<ea>
EOR.B #0F45Bh,D4	EORI.B #0F45Bh,D4

Automatic Substitution of MOVE Instructions

The Assembler can substitute specialized versions of the MOVE instruction as follows:

IF YOU SPECIFY:	THE ASSEMBLER PRODUCES:
Normal Move	
MOVE <ea>,<ea>	MOVE <ea>,<ea>
MOVE D7,(A4)	MOVE D7,(A4)
Move Address	
MOVE <ea>,An	MOVEA <ea>,An
MOVE TABLE,A1	MOVEA TABLE,A1
Move Quick	
MOVE #data,Dn	MOVEQ #data,Dn
MOVE #345,D0	MOVEQ #345,D0

Automatic Substitution of OR Instructions

The Assembler can substitute specialized versions of the OR instruction as follows:

IF YOU SPECIFY:	THE ASSEMBLER PRODUCES:
Normal OR	
OR <ea>,Dn	OR <ea>,DN
OR D4,D5	OR D4,D5
OR Dn,<ea>	OR Dn,<ea>
OR D3,(A3)	OR D3,(A3)
OR Immediate	
OR #data,<ea>	ORI #data,<ea>
OR.B #100B,(A1)	ORI.B #100B,(A1)

Automatic Substitution of Subtract Instructions

The Assembler can substitute specialized versions of the SUB instruction as follows:

IF YOU SPECIFY:	THE ASSEMBLER PRODUCES:
Normal Subtract	
SUB <ea>,Dn	SUB <ea>,Dn
SUB D1,D2	SUB D1,D2
SUB Dn,<ea>	SUB Dn,<ea>
SUB D3,(A5)	SUB D3,(A5)
Subtract Address	
SUB <ea>,An	SUBA <ea>,An
SUB COUNT,A1	SUBA COUNT,A1
Subtract Immediate	
SUB #data,<ea>	SUBI #data,<ea>
SUB #50,D2	SUBI #50,D2
Subtract Quick	
SUB #data,<ea>	SUBQ #data,<ea>
SUB #3,D4	SUBQ #3,D4

Because the Assembler cannot automatically substitute the SUBX instruction for the SUB instruction, you must explicitly use the SUBX form when you want to perform a subtract with extend.

THE OPERAND FIELD

The operand field provides data that is to be used either by an instruction or by a pseudo operation code. Operands can be:

- The name of any 68000 register.
- Any label defined in the module or declared as a global label. (A global label is a label that can be used in any module of a program regardless of the module in which it is defined.)
- Any legal constant as described under the heading Constants.
- Any legal expression as described under the heading Expressions.
- Any effective address as described under the heading Effective Addresses.

Each instruction and pseudo operation code requires that specific kinds of data be provided in its operand field. Names of macros may not be used as operands; instead, they are used as operation codes, and the Assembler substitutes the correct code at assembly time. Operands for some pseudo operation codes, such as TITLE and *INCLUDE, are not operands in the sense described here and are subject to other restrictions.

Undefined expressions for the IF, EQU, DL, ORG, REPT, STRUCT, and DS pseudo operation codes give errors on pass 1 of an assembly. This avoids the generation of a complex phase error during pass 2.

Labels Used as Operands

When you use labels as operands, the Assembler substitutes the label's value for the label in the statement. If you have these statements in a module:

```
LOOP:       . . .
              :
              :
            BRA     LOOP
```

the Assembler will substitute the address of the statement that defines **LOOP** for the word **LOOP** in the BRA statement.

If another module defines the value for a label, you must define that label as a global label. Then the Linker will substitute the correct value for the label when it links the modules of the program together. If you do not declare a label to be global (Chapter 4 describes how this is done) and the Assembler does not find a definition for the label in the module, it marks any use of the label as an operand as an error.

Through the use of the EQU and SET pseudo operation codes (see Chapter 5 for details), you can give a label any value. The following example uses register D3 as a counter. To make the code more readable, the programmer uses EQU to give the label **COUNTER** the value of D3:

```
COUNTER  EQU     D3
```

Now the programmer can write statements whose functions are obvious:

```
ADD      #1,COUNTER
```

instead of the more cryptic:

```
ADD      #1,D3
```

When using labels, be sure to use the appropriate syntax. For instance, if you equate **TABLE** to register A2, this statement:

```
MOVE     D2,TABLE
```

would move the value from register D2 into register A2, while this statement:

```
MOVE     D2,(TABLE)
```

would move the value from register D2 to the location in memory specified by the value in register A2.

Constants Used as Operands

The Assembler allows binary, octal, hexadecimal, decimal, and ASCII constants according to the following conventions:

Decimal Numbers formed from decimal digits (0-9) and either left unterminated or terminated by the character "D".

Example: MOVE #11130,D1

Binary Numbers formed from binary digits (0,1) and terminated by the character "B".

Example: MOVE #10101101111010B,D1

Octal Numbers formed from octal digits (0-7) and terminated by the character "Q".

Example: MOVE #25572Q,D1

Hex Numbers formed from hexadecimal digits (0-9 and A-F) and terminated by the character "h". A hex number beginning with a letter must be preceded by a "0" to distinguish it from a label or register name.

Example: MOVE #2B7Ah,D1

ASCII Numbers represented by ASCII characters are enclosed in single quotes. Single quotes are represented by two single quote characters ('').

Example: MOVE #'+z',D1

ASCII constants may be longer than 4 characters (32 bits). The Assembler will keep all the characters if possible. If a 32-bit quantity is required, the Assembler will truncate a string to the last four characters.

```
Examples:

DC.B   'ABCDEF'   ;      gives  414243444546

DC     'ABCDEF'+1 ;      gives  43444547
```

A pound sign (#) precedes constants' values so that the Assembler recognizes the value as a constant and not as a label. Each of the previous examples produces the **same value** in the D1 register upon assembly and execution.

Expressions Used as Operands

An expression is any operand that the Assembler must evaluate to determine a value that it is to substitute for that operand. In this statement:

```
MOVE     #TABLE+100h,A3
```

the Assembler adds 100h to the value for the label **TABLE** and then moves the sum into register A3. The pound sign (#) preceding the expression tells the Assembler to treat the result as a numeric value. (Or, more technically, it tells the Assembler to treat the expression as immediate data. See Immediate Data in this chapter for more information on this form of addressing.) The expression is legal without the pound sign:

```
MOVE     TABLE+100h,A3
```

Now, however, the Assembler places the value found at the location **TABLE+100h** in memory into register A3.

In this statement:

```
IF     COUNT>TOTAL
```

the Assembler determines if the value for **COUNT** is greater than the value for **TOTAL** and then takes the appropriate action. You can use expressions in place of either address or constant operands provided they do not evaluate to an illegal quantity.

If the result of an expression is 0, it is false; if the result of an expression is other than 0 (specifically, -1), it is true. These expressions are all true:

```
        IF      1+1

LABEL   EQU     1000h
        IF      LABEL

        IF      2 > 1

        IF      'abcd' > 'aaaa'
```

These expressions are all false and give a value of zero:

```
        IF      0

LABEL   EQU     0
        IF      LABEL

        IF      1 > 2

        IF      1-1

        IF      'abcde' = 'abcdef'
```

See Chapter 3 for more information on IF.

You can use relational expressions (e.g., equal to, greater than, less than) to specify values for instructions. This example:

```
ADD     #2>1,D2
```

adds the value -1 (the result of evaluating 2>1) to whatever value register D2 holds.

Using Operators in Expressions - The following operators may be used to form expressions:

+	Addition or Positive - binary or unary
-	Subtraction or Negative - binary or unary
*	Multiplication

/	Division
**	Exponentiation
% or MOD	Modulus - Compute the remainder of a division. X MOD Y is defined to be X-(Y*INT(X/Y)). If X=23 and Y=7 then X MOD Y=2. (**INT** is the largest integer value that is less than or equal to the expression.)
> or GT	Greater Than - True if the left operand is greater than the right operand.
>= or GE	Greater Than or Equal - True if the left operand is greater than or equal to the right operand.
< or LT	Less Than - True if the left operand is less than the right operand.
<= or LE	Less Than or Equal - True if the left operand is less than or equal to the right operand.
= or EQ	Equals - True if the left and right operands are equal.
<> or NE	Not Equal - True if the left and right operands are not equal.
<< or SHL	Shift Left Logical - Shift n places. If X=2Ah, then X SHL 1=54h.
>> or SHR	Shift RIGHT Logical - Shift n places If X=2Ah, then X SHR 2=0Ah.
~ or NOT	Logical NOT - Unary
& or AND	Logical AND - If X=C0h and Y=47h, then X AND Y=40h.
| or OR	Logical OR - If X=C0h and Y=47h, then X OR Y=C7h.
XOR	Exclusive OR - If X=C0h and Y=47h, then X XOR Y=87h.
^	Set bit as specified by the expression following the operator. This is a unary operator and may alter bit 0-31 to form an integer constant. This operator has the highest precedence.

Note: At present, keywords for operators such as AND or OR are not implemented.

Unless otherwise stated, all operators perform binary operations.

Operators written as symbols (e.g., **+**) can be separated from their operands by a SPACE. Operators written as one or more letters (e.g., **GE**) must be separated from their operands by a SPACE.

The Assembler considers these operators to have a hierarchy that determines which take precedence over others. The list in Table 2-3 gives this hierarchy, progressing downward from those of highest priority to those of lowest priority; all those operations on the same level are of equal priority. Operators that are on the same level of the hierarchy would be evaluated from left to right as they occur in an expression. Operators or parts of expressions enclosed in parentheses or brackets are evaluated first, beginning with the innermost set. Operators of the same level are executed in the order in which they occur.

Table 2-3: HIERARCHY OF OPERATORS

Hierarchical Level	Operator	Description
1	**	Exponentiation
2	+	Positive number
	-	Negative number
	~	Logical NOT
	^	Bit
3	*	Multiplication
	/	Division
	%	Modulus
	<<	Shift left logical
	>>	Shift right logical
4	+	Addition
	-	Subtraction
5	&	Logical AND
6	|	Logical OR
	XOR	Exclusive OR
7	<	Less than
	>	Greater than
	=	Equal to
	<>	Not equal to
	<=	Less than or equal to
	>=	Greater than or equal to

Comparing ASCII Strings - An ASCII string comparison is of the following format:

"string-1" relational-operator "string-2"

where the relational operator can be any one of:

Operator	Description
EQ (=)	Equal
NE (<>)	Not equal
GT (>)	Greater than
GE (>=)	Greater than or equal
LT (<)	Less than
LE (<=)	Less than or equal

and string-1 and string-2 may be of any length. If one string is shorter than the other, the shorter is left-justified and padded with nulls. An example of a string comparison would be:

```
IF      '\PARAMETER' EQ 'YES'
```

Current Program Counter ($) - The dollar sign ($) may be used in the operand of any operation code allowing expressions as operands. The dollar sign is used to represent the current program counter of the Assembler. Note that dollar sign points to the beginning of the statement that contains it and not to the end. An example of the way to use it is:

```
DATA:     DC.B     0,11,3,2,7,24,17
COUNT:    EQU      $-DATA
```

The name **COUNT** has the value of 7, because this is the number of entries in **DATA** (the address of DATA subtracted from the current location). Elsewhere in the source program, **COUNT** can be used to stand for the number of entries in **DATA.** There is great advantage to this representation; if it becomes necessary to change the number of entries of **DATA** and reassemble, the value of **COUNT** is changed automatically. If an absolute 7 were used instead of **COUNT,** every occurrence of the 7 in the source program would have to be changed.

The dollar sign is often used in another way that is actually poor programming practice. That is to use the dollar sign in a relative jump instruction. The best way to handle relative jumps is to label the location to be jumped to, and use this label as the operand of the jump instruction. The Assembler then calculates the correct displacement.

Effective Addresses Used as Operands

Many instructions specify the location of one or more operands through the use of effective addresses, which are expressions defining locations. The Assembler supports the full range of effective address modes allowed in the 68000 assembly language. The following sections define the syntax of the different modes and give examples of each. Table 2-4 summarizes the different modes. (Cromemco's Assembler uses the same syntax as the Motorola Assembler.)

Table 2-4: SUMMARY OF EFFECTIVE ADDRESS MODES

Effective Address Mode	Description
Data Register Direct	Syntax: Dn Example: MOVE D0,D5 Addressing Categories: Data, Alterable
Address Register Direct	Syntax: An Example: MOVEA D0,A5 Addressing Categories: Alterable
Address Register Indirect	Syntax: (An) Example: MOVE D0,(A5) Addressing Categories: Data, Memory Control, Alterable
Address Register Indirect with Postincrement	Syntax: (An)+ Example: MOVE D0,(A5)+ Addressing Categories: Data, Memory, Alterable
Address Register Indirect with Predecrement	Syntax: -(An) Example: MOVE D0,-(A5) Addressing Categories: Data, Memory, Alterable
Address Register Indirect with Displacement	Syntax: d(An) Example: MOVE D0,4(A5) Addressing Categories: Data, Memory Control, Alterable

Address Register Indirect with Index

Syntax: d(An,Rm.W)
d(An,Rm.L)
Examples: MOVE D0,4(A5,D1)
MOVE D0,4(A5,D1.L)
Addressing Categories:
Data, Memory
Control, Alterable

Immediate Data

Syntax: #expression
Example: MOVE #33+COUNT,D0
Addressing Categories:
Data, Memory

Program Counter with Displacement

Syntax: label
Example: MOVE D0,COUNTER
Addressing Categories:
Data, Memory, Alterable

Program Counter with Index

Syntax: label(Rm.W)
label(Rm.L)
Examples: MOVE D0,COUNTER(D5)
MOVE D0,COUNTER(D5.L)
Addressing Categories:
Data, Memory, Alterable

Absolute Short Address

Syntax: label
Example: MOVE D0,ADDRESS.W
Addressing Categories:
Data, Memory
Control, Alterable

Absolute Long Address

Syntax: label
Example: MOVE D0,ADDRESS.L
Addressing Categories:
Data, Memory
Control, Alterable

Data Register Direct - In this mode, the effective address is a data register. The syntax is **Dn**, where **n** is the number of the data register.

Syntax: Dn
Example: ADD #10h,D5

Before execution:

Register D5 contains 3400

Changed values after execution:

Register D5 contains 3410

This example adds the value 10h to the value in register D5.

Address Register Direct - In this mode, the effective address is an address register. Note that the register itself is the address, not the location in memory it specifies. The syntax of this form is **An**, where **n** is the number of the address register.

Syntax: An
Example: ADD #10h,A2

Before execution:

Register A2 contains 00004000

Changed values after execution:

Register A2 contains 00004010

This example adds the value 10h to the value in register A2.

Address Register Indirect - In this mode, the effective address is the location in memory specified by the value in an address register. Note that it is the memory location that is the address, not the register itself. The syntax of this form is **(An)**, where **n** is the number of the address register.

Syntax: (An)
Example: ADD D1,(A3)

Before execution:

Register D1 contains	10
Register A3 contains	00004000
Location 00004000 contains	3821

Changed values after execution:

Location 00004000 contains	3831

This example adds the value in register D1, 10h, to the value at location 00004000, 3821, with the result that location 00004000 contains the value 3831 after the execution of this statement.

Address Register Indirect with Post-increment - In this mode, the effective address is the location in memory specified by the value in an address register. Once the instruction is executed, the value in the address register is incremented by 1 if the operand size is a byte, by 2 if the operand size is a word, and by 4 if the operand size is a long word. When the stack pointer register is the address register and the operand size is a byte, the value is incremented by 2 so that the stack continues to point to a word boundary.

The syntax of this form is **(An)+** where **n** is the number of the address register.

Syntax: (An)+
Example: ADD #3,(A3)+

Before execution:

Register A3 contains	00004000
Location 00004000 contains	3821

Changed values after execution:

Register A3 contains	00004002
Location 00004000 contains	3824

This example adds the value 3 to the value at location 00004000, 3821, with the result that location 00004000 contains the value 3824 after the execution of this statement. The value in register A3, 00004000, is incremented by 2 because the operand size is a word.

Address Register Indirect with Pre-decrement - In this mode, the effective address is the location in memory specified by the value in an address register minus a value equal to the length of the operand. Before the instruction is executed, the value in the address register is decremented by 1 if the operand size is a byte, by 2 if the operand size is a word, and by 4 if the operand size is a long word. When the stack pointer register is the address register and the operand size is a byte, the value is decremented by 2 so that the stack continues to point to a word boundary.

The syntax of this form is **-(An)** where **n** is the number of the address register.

Syntax: -(An)
Example: ADDQ #3,-(A3)

Before execution:

Register A3 contains	00004000
Location 00003FFE contains	3825

Changed values after execution:

Register A3 contains	00003FFD
Location 00003FFE contains	3828

This example decrements the value of register A3, 00004000, by 2 because the operand size is a word. It then adds the value 3 to the value at location 00003FFE (the result of 00004000-2), 3825, with the result that location 00003FFE contains the value 3828 after the execution of this statement.

Address Register Indirect with Displacement - In this mode, the effective address is the location in memory specified by the value in an address register plus a displacement. The syntax of this form is **d(An)** where **d** is the displacement and **n** is the number of the address register.

Syntax: d(An)
Example: ADDQ #3,100h(A3)

Before execution:

Register A3 contains	00004000
Location 00004100 contains	3825

Changed values after execution:

Location 00004100 contains	3828

This example adds the value of register A3, 00004000, with the displacement, 100h. It then adds the value 3 to the value at location 00004100 (the result of 00004000+100), 3825, with the result that location 00004100 contains the value 3828 after the execution of this statement.

Address Register Indirect with Index - In this mode, the effective address is the location in memory specified by the value in an address register plus a displacement and an index value. The syntax of this form is **d(An,Rm.W)** or **d(An,Rm.L)** where **d** is the displacement and **n** is the number of the address register, **Rm** is either a data or address register, **W** means the index value is 2 bytes long, and **L** means the index value is 4 bytes long. If the size of the index is not specified, it is assumed to be word (.W).

Syntax: d(An,Rm.W)
d(An,Rm.L)
Example: ADDQ #3,10h(A3,D2)

Before execution:

Register A3 contains	00004000
Register D2 contains	00000050
Location 00004060 contains	3825

Changed values after execution:

Location 00004060 contains	3828

This example adds the value of register A3, 00004000, with the displacement, 10h and the value of the index register, 50h. It then adds the value 3 to the value at location 00004150 (the result of 00004000+100+50), 3825,

with the result that location 00003FFD contains the value 3828 after the execution of this statement.

Immediate Data - In this mode, the effective address is an expression in the statement. The syntax of this form is **#expression** where the expression can be any legal expression and can be 1, 2, or 4 bytes long.

Syntax: Dn
Example: ADDI #5+COUNT,D5

Before execution:

COUNT has the value	0005
Register D5 contains	3400

Changed values after execution:

Register D5 contains	340A

This example adds the value 0Ah (the result of 5+5) to the value in register D5.

Program Counter with Displacement - In this mode, the effective address is the location specified by an expression or label. The Assembler calculates the difference between the current program counter and the specified location and uses the difference as a displacement.

Syntax: label or expression
Example: ADD COUNTER,D5

Before execution:

COUNTER has the value	0100
Location 0100h contains	0004
Register D5 contains	0005

Changed values after execution:

Register D5 contains	0009

This example adds the value at the location specified by COUNTER, 4 to the value in register D4, 5, with the result that register D5 contains the value 9 after the execution of this statement.

Program Counter with Index - In this mode, the effective address is the sum of the address in the program counter, the given (8-bit) displacement and the contents of the index register.

The Assembler calculates the difference between the current program counter and the specified location and uses the difference as a displacement. The syntax of this form is **d(Rm.W)** or **(Rm.L)** where **d** is the displacement, **Rm** is either a data or address register, **W** means the value is 2 bytes long, and **L** means the value is 4 bytes long.

Syntax: d(Rm.W)
d(Rm.L)
Example: ADD COUNTER+10h(A2),D5

Before execution:

COUNTER has the value	0100
Register A2 has the value	1000
Location 1110h contains	0004
Register D5 contains	0005

Changed values after execution:

Register D5 contains	0009

This example adds the value at the location specified by COUNTER, 4, plus 10h, plus the value in register A2, 1000h, to the value in register D4, 5, with the result that register D5 contains the value 9 after the execution of this statement.

Absolute Short Address - In this mode, the effective address is the absolute location in memory specified by the value of a label or expression. In the absolute short address mode, the value is 2 bytes long.

Syntax: label or expression
Example: ADD COUNTER.W,D5

Before execution:

COUNTER has the value	1000
Location 1000h contains	0004
Register D5 contains	0005

Changed values after execution:

Register D5 contains	0009

This example adds the value at location 1000h, 4, to the value in register D5, 5, with the result that register D5 contains the value 9 after the execution of this statement.

Absolute Long Address - In this mode, the effective address is the absolute location in memory specified by the value of a label or expression. In the absolute long address mode, the value is 4 bytes long.

Syntax: label or expression
Example: ADD COUNTER.L,D5

Before execution:

COUNTER has the value	10000000
Location 10000000h contains	0004
Register D5 contains	0005

Changed values after execution:

Register D5 contains	0009

This example adds the value at location 10000000h, 4, to the value in register D5, 5, with the result that register D5 contains the value 9 after the execution of this statement.

Choosing the Addressing Mode

When the Assembler finds a label in the position of the effective address, e.g., in the instruction

```
CLR     LABEL
```

it must decide what actual form the effective address should take. The various possibilities are:

absolute long address
absolute short address
program counter relative address
address register with displacement

For a detailed description of the fourth possibility, see the description of **USAGE** and **DROP** pseudo operation codes.

It is not obvious, to the Assembler or to the programmer, which one is best. This section contains a discussion of the advantages and drawbacks to the use of each addressing modes.

Absolute long addresses can always be used, but each occupies 4 bytes.

Absolute short addresses are limited to 32K bytes in the program. If you know that the complete program will reside within memory addresses 000000h - 007FFFh, then the use of absolute short addresses is certainly a good choice. Currently, no user program can occupy these locations. Actual addresses of the user program will be 020000h or higher, which rules out absolute short addresses altogether.

Program counter (PC) relative addressing seems to be the best, though it has two drawbacks. First, the range is again limited to 32K bytes. A more serious drawback is that many instructions do not allow PC relative addressing. (PC addressing is supposed to apply only to a read-only part of the memory; therefore each 68000 instruction that writes into memory does not allow PC relative addressing to be used).

Address register with displacement seems to be very convenient, except for the fact that:

1. At least one A register has to be set aside.

2. The programmer must issue an appropriate USING instruction.

3. The programmer is responsible for the loading of the designated A register with the correct information.

This means that A register with displacement addressing is not so convenient at all.

This discussion leads to the conclusion that the Assembler simply cannot be smart enough to make the best choice. The programmer must help to make the most critical decisions by setting a number of address mode selection bits in the Assembler. These bits govern the decision process of choosing the appropriate effective address form. This section lists the meaning of those bits.

Bit ABS_L Specifies the type of absolute address.

0 means all absolute addresses will be short (2 bytes).
1 means all absolute addresses will be long (4 bytes).

As has been discussed, 1 is the necessary choice with the current operating system.

Bit EXT_PC Specifies that PC relative addressing is allowed for external symbols.

0 means PC relative addressing is not allowed for external symbols.
1 means PC relative addressing is allowed for external symbols.

In general, PC relative addressing should be allowed for external symbols. If the instruction does not allow it, the Assembler will not use it. However, if you plan to create a program larger than 32K bytes, the Linker may not be able to link the program because of a reference that exceeds the range of PC relative addressing. If so, the Linker will print an error message, and the resulting bin file will not be executable.

Bit EXT_ABS Specifies that absolute addressing is allowed for external symbols.

0 means external references should not be translated into absolute addresses.
1 means external references may be translated into absolute addresses.

Remember there are two absolute addressing modes, short and long, as previously discussed under ABS_L.

At first, bit EXT_ABS seems to be the complement of the bit EXT_PC: If PC relative addressing should not be used, then the Assembler must use absolute addressing, and vice versa. However, this is not true. There is also address register with displacement as a possibility and the meaning of the bit is what it says: if absolute addresses are allowed, the Assembler may, but not must, use them. If disallowed, absolute addresses will not be used and the Assembler will use another addressing mode if one is available. If there is no choice, the Assembler will generate the ADDRESS MODE error.

Bit OTH_PC Specifies that PC relative addressing may be used for the symbols in some other PSECT.

0 means PC relative addressing may not be used for symbols in another PSECT.

1 means PC relative addressing may be used for symbols in another PSECT.

This bit does the same for symbols in another PSECT as EXT_PC does for external symbols.

Bit OTH_ABS Specifies that absolute addressing may be used for symbols in another PSECT.

0 means absolute addressing may not be used for symbols in another PSECT.

1 means absolute addressing may be used for symbols in another PSECT.

This bit does the same for symbols in another PSECT as EXT_ABS does for external symbols.

Bit FWD_L Specifies that forward references must be long absolute addresses.

0 means forward references need not be (but may be) long (absolute addresses).

1 means forward references must be long absolute addresses.

First of all, this bit has an effect only in the case that ABS_L = 1, i.e., long absolute addresses are selected. In general, this bit is not set. You can set it if you want to make the Assembler foolproof in the selection of addressing modes. If all forward references assemble into long absolute addresses, the Assembler will not discover in the second pass that it cannot assemble the program.

Bit FWD_S Specifies that forward references must be 2-byte addresses.

0 means forward references may be (but need not be) 2-byte addresses.

1 means forward references must be 2-byte addresses.

This bit becomes effective only if ABS_L = 1. In general, this bit is not set. It should be set only if the Assembler can use another addressing mode, e.g., PC relative or address register with displacement.

The overview of the FWD bits follows:

FWD_L	FWD_S	
0	0	Assembler will use PC relative addressing if the instruction permits; otherwise, it will use absolute long addresses.
0	1	Assembler will choose a short address in the first pass. In the second pass, the Assembler assumes that either PC relative or address register mode with displacement will work. If the Assembler cannot use PC relative or address register mode, it will display the ADDRESS MODE error. This combination produces the most compact code, where short addresses suffice for all references.
1	0	All forward references will be assembled into long absolute addresses. This is the safest (and the most expensive) solution.

It should be stressed that the preceding discussion is meaningless if you have selected short absolute addresses. In this case, all addressing modes use 2-byte addresses and forward references present no special problem.

Bit ABS_ALLOW Specifies that absolute (i.e., not relocatable) addresses are allowed.

0 Nonrelocatable addresses are not allowed.

1 Nonrelocatable addresses are allowed.

The so-called absolute addresses (short and long) are still relocatable in the sense that the final value is computed by the linker and the operating system program loader. A nonrelocatable address is a number or a label equated to such a number. In general, there is no need to use such addresses because the program does not have to know where in memory any of the information is actually stored. Nonrelocatable addresses usually result from a programming error, for example:

```
MOVE    45,D0
```

instead of

```
MOVE    #45,D0
```

There are rare instances where a nonrelocatable address is required, for example:

```
MOVE.L chk_routine, 18h ;18h is CHK trap address
```

All such instructions, however, interfere with the CROMIX Operating System, and it is therefore preferable to disable nonrelocatable addresses. They may be enabled by setting bit ABS_ALLOW. The effect of this bit may be overridden by writing **.W** or **.L** after an address, for example:

```
MOVE.L          chk_routine, 0018h.W
```

The address 18h will be assembled into absolute word (for .W) address, which is nonrelocatable regardless of the settings of any bits.

These bits are stored in a 2-byte word in the Assembler. There is a special pseudo instruction, OPT, with the syntax

```
OPT     <value>
```

which sets this word to an arbitrary combination of bits. To simplify programming, an include file **optequ.asm** on the distribution diskette defines the numbers of these bits, including the standard setting:

```
OPT_DEFAULT    equ    ^ABS_L | ^EXT_ABS | ^OTH_ABS
```

so that, for example, restoring to this combination may be achieved by

```
OPT     OPT_DEFAULT
```

Similarly, you can use

```
OPT     OPT_DEFAULT | ^FWD_S
```

to prevent long forward references.

THE COMMENT FIELD

The comment field can contain anything you want, although it is typically used to explain the execution of the program. The field is free-format and can include any printable ASCII characters, as long as the comment is preceded by a semicolon (;). The comment may follow an operation code, operand, or label or may exist on a line by itself. The semicolon must be the first non-blank or non-TAB character on the line for the comment to be on the line by itself. Multiple blanks or TABs may be used before or within the comment to improve readability. A RETURN terminates the comment. Comments may appear on any line except those that have the pseudo operation codes TITLE, SUBTTL, EJECT, or FORM as the operation.

Chapter 3

MACROS, CONDITIONAL ASSEMBLY, AND REPEAT EXPANSIONS

The three facilities described in this chapter provide much of the Assembler's power. They make your code both simpler to write and more flexible:

-- Macros and repeat expansions simplify the writing. They allow you to define a block of code once and then have that block appear any number of times in a module. However, the two facilities do differ. The Assembler can repeat a macro anywhere within a module. The Assembler can repeat a repeat expansion only at the place you define it.

-- Macros, conditional assembly, and repeat expansions make the code more flexible. Conditional assembly allows you to specify the conditions under which the Assembler includes a particular code block in the module. Macros and repeat expansions allow you to vary the blocks of code. Each time they appear, they can be tailored to a specific situation.

MACROS

Macros allow you to define a block of code that the Assembler can insert anywhere in a module. They give you more flexibility than in-line source code because you can modify their code each time you use them with parameters. Suppose, for example, you frequently need to move blocks of 100 and 500 bytes. You could rewrite the statements for the moves each time you need them. Writing a single macro that accepts the appropriate parameters, however, saves time.

Macros have other advantages:

- You can use macros to make your modules more readable.

- You can create libraries of macros that anyone can use.

- You can change the functions of the 68000 instruction mnemonics.

- You need to debug a macro once, no matter how many times it is used.

While macros provide flexibility and power, use them with a certain amount of restraint. As with the GOTO statement of high-level languages, excessive and inconsistent use of macros can make a program difficult to read and modify.

When several programmers work on the same project, it is wise to have everyone use a standard macro library. This way each programmer writes code that can be easily read and modified by the other programmers.

At assembly time the macro expands and the source code generated is printed on consecutive lines following the macro call pseudo operation code (unless NOGEN is selected--see Assembler call options in Chapter 8 and the pseudo operation code descriptions in Chapter 7). Each of these lines has a plus sign (+) immediately following the line number of the print listing to distinguish these lines as belonging to a macro expansion.

Macros Versus Subroutines

You may wonder how macros differ from subroutines, since subroutines may also be used to reduce the coding of frequently executed blocks of code. One distinction is that subroutines call other parts of the program, while macros generate in-line code. However, a macro does not necessarily generate the same source code each time it is called. The source code the macro generates can be changed by changing the parameters in the macro call. Also, macro parameters can be tested at assembly time by the conditional assembly (IF) construction. These two features enable a general-purpose macro definition to generate customized source code for a particular situation.

The biggest difference between macros and subroutines is that macros can produce customized in-line code. Subroutines, on the other hand, reside in the source program and require extra execution time (especially if the subroutines perform any conditional operations).

There is a trade-off, however, between the extra memory required for macros (in-line code) and the longer execution time of subroutines. In most cases, using a single subroutine rather than multiple in-line macros reduces the overall program size. However, the use of macros may be more efficient in situations involving a large number of parameters. Note that macros can call subroutines, and subroutines can contain macro calls.

Sample Macros

An example of a simple macro definition illustrates some of the features of macros. Suppose you wanted to shift the bits in a long word four bits to the right a number of times in a source module. You could write a macro to do this that has a name that clearly specifies the function to be done:

```
SHIFT:  MACRO
        LSR.L   #4,D0
        MEND
```

The general format of a macro definition can be seen from this example. The word **SHIFT** is the macro name. To call this macro, simply use the word **SHIFT** as an operation code in the source code. The Assembler inserts the LSR operation code as in-line source code following the **SHIFT** macro operation code. This process is known as the macro expansion. The MEND statement informs the Assembler that the macro definition is complete.

Suppose now that, rather than having the macro shift only the D0 register, you want it to operate on any of the data registers. The following defines such a macro:

```
SHIFT:  MACRO   \REGIS
        LSR.L   #4,\REGIS
        MEND
```

This macro uses the parameter **REGIS**, the value of which the Assembler determines when the SHIFT macro is called. The backslash symbol (\) precedes the parameter in the macro definition to distinguish it from other, fixed, parts of the definition.

Since SHIFT now expects one parameter, the call for the macro is:

```
        SHIFT   register
```

where the word **register** is replaced with the name of the register you want to shift. Upon finding this call, the Assembler generates in-line code using the correct register name. For example, if the macro call

```
        SHIFT   D4
```

were used, the Assembler would generate the in-line code:

```
        LSR.L   #4,D4
```

The next example shows a macro that moves a block of data:

```
BMOVE:  MACRO   \SOURCE,\SRCEND,\DESTIN
        LEA     \SOURCE,A0
        LEA     \DESTIN,A1
        MOVE    #\SRCEND-SOURCE-1,D0
AA\SYM: MOVE.B  (A0)+,(A1)+
        DBRA    D0,AA\SYM
        MEND
```

Three parameters are expected: 1) a starting location for the source; 2) an ending location for the source; and 3) a destination. The macro call for this example might be part of a module with code such as:

```
                :
                :
INIT:       DC      0,1,2,3,4,5
INITEND:
DATA_AREA:  DS.B    INITEND-INIT
```

To perform a block move that would initialize **DATA_AREA** with the values stored between **INIT** and **INITEND**, you would use the call:

```
BMOVE       INIT,INITEND,DATA_AREA
```

which would produce the code:

```
            LEA         INIT,A0
            LEA         DATA_AREA,A1
            MOVE        #INITEND-INIT-1,D0
AA000000:   MOVE.B      (A0)+,(A1)+
            DBRA        D0,AA000000
```

The Assembler produces the label **AA000000** through a special feature that generates a unique label each time the macro is called. Defining Labels in Macro Definitions in this chapter describes this feature in more detail.

The Assembler allows you to define macros in either the source file or in special macro definition libraries. Any macros you define in a source file must appear before the first call to that definition.

To use a macro library, you must write the ***MACLIB** pseudo instruction (see Chapter 7). With one important exception, the result is almost the same as if you simply ***INCLUDE** the macro library. If a macro definition is included in the source file, the Assembler stores the macro body into memory. With ***MACLIB** only the name of the macro is stored; the body is stored only if the macro is actually called.

Note: Macro libraries are not yet implemented.

Writing the Macro Definition

A macro definition follows this format:

```
name:    MACRO[.\param0]  [\param1,\param2,...]
                :
           macro body
                :
(no label) MEND (no operand)
```

where:

name:

is the name of the macro and must be given. Use the name to call the macro. (See Writing the Macro Call in this chapter.)

The syntax rules applying to macro names are the same rules applying to writing labels in general. The name can be of any length and may consist of any letters, any numerals, the dollar sign, the underscore symbol, the question mark, and the at sign (@). A colon, SPACE, or TAB separates the name from the MACRO pseudo operation code.

MACRO

is the MACRO pseudo operation code.

The macro name and the parameters appear on the macro statement.

\param0

is an optional parameter that the Assembler substitutes with the string given as the size extension in the macro call. The backslash indicates that this is a parameter and must precede the parameter name everywhere it appears in the macro definition. The name can be any length that fits on the statement. Any printable character can appear in the name. Only the preceding period and backslash separates the parameter name from the MACRO pseudo operation code.

Parameters in this chapter gives more information on the use of parameters.

\param1,\param2,...

are optional "dummy" names that the Assembler substitutes with values given in the macro call. As many parameters can appear as will fit on the MACRO statement.

The Assembler treats the parameters as the MACRO statement's operands. The rules given in Chapter 2 for placing the operand on a statement apply to the parameter names.

The backslash indicates that these are parameters and must precede the parameter names everywhere they appear in the macro definition. The name can be any length that fits on the statement. Any printable character can appear in the name. Commas separate the parameter names.

Parameters in this chapter gives more information on the use of parameters.

macro body

are the instructions and pseudo operation codes that the Assembler substitutes for the macro call. You can use any legal 68000 instruction or pseudo operation code within the macro body. The macro body can be any length. If you include parameters within the body, the Assembler substitutes them with values given in the macro call.

MEND

is the required MEND pseudo operation code that marks the end of the macro definition.

The following sections describe various features of the macro definition in more detail.

Specifying Labels in Macro Definitions - A standard label appearing in a macro definition would generate a multiple definition error if you call that macro more than once. Each expansion of the macro would produce the same label, an illegal situation. (Labels appearing on SET pseudo operation codes within the macro definition are not subject to the above restriction because they can be multiply defined in the same module.) To avoid this problem, the Assembler provides a general label name for macro definitions that is used by assigning a short name to the label name followed by the characters **\SYM.** Each time the source code calls the macro, the Assembler replaces the \SYM with a six-digit number.

For example, the "dummy" label name AA\SYM in this macro:

```
BITEST:   MACRO   ...
            :
            :
AA\SYM:   LEA     A5,...
            :
            :
          BRA     AA\SYM
```

would be assigned the actual label name AA000000 if BITEST is the first macro called in the module. The expanded macro would then look like this:

```
            :
            :
AA000000: LEA     A5,...
            :
            :
          BRA     AA000000
```

The six-digit number starts at 000000 and is incremented by one each time any macro is called, whether or not it is a macro with a \SYM label. For the first macro call in the source file, the \SYM would be replaced with 000000, the second call would produce 000001, the third, 000002, and so on.

In general, do not use \SYM as the name of a parameter in a macro definition. If you do, the current value of \SYM is used instead of the desired parameter.

Using Parameters - Parameters give macros their power. With them, each use of the definition can produce unique code tailored to the needs of a given situation.

Within a macro definition, "dummy" names represent the parameters. When the Assembler expands the macro call, it will take values given for the parameters in the call and substitute the values for their corresponding names in the definition.

Each parameter used in the definition must be listed as an operand of the MACRO statement. The Assembler assigns values given as operands of the macro call to the parameters in the order in which they appear.

If you do not give a value for one or more of the parameters, the Assembler substitutes a null value for the parameter names. In this example:

```
EXAMPLE:  MACRO     \ADDRESS_REG,\DISPLACEMENT,\DATA_REG
          ADD       \DISPLACEMENT(ADDRESS_REG),\DATA_REG
            :
            :
          MEND
```

you can skip the displacement value as needed by just not giving a value for the displacement parameter as is done here:

```
EXAMPLE A2,,D3
```

When you want to skip a parameter in the beginning or middle of a series of parameters, you must use commas to indicate the omitted parameters. When you want to skip a parameter at the end of a series, you can either use commas to indicate the parameters or end the parameters with the last parameter that is to have a value.

If a macro has three parameters, you would use these calls to selectively skip each of the parameters:

```
First parameter:    EXAMPLE  ,20,D3

Second parameter:   EXAMPLE  A1,,D3

Third parameter:    EXAMPLE  A1,20, or
                    EXAMPLE  A1,20
```

When the Assembler expands a macro definition, it first substitutes the values for the parameters to generate new source code statements. It then produces object code from the expanded source code. You must ensure that the values for the parameters have the proper syntax for their use. For example, this macro:

```
EXAMPLE:  MACRO     \ADDRESS_REGISTER
          ADD       #10,\ADDRESS_REGISTER
          MEND
```

could be used to add 10 to the value in an address register with this call:

```
EXAMPLE A4
```

or could be used to add 10 to the value of the location specified by an address register with this call:

```
EXAMPLE (A4)
```

If the macro call is written with parentheses surrounding the parameter name on the ADD statement:

```
EXAMPLE:  MACRO     \ADDRESS_REGISTER
          ADD       #10,(\ADDRESS_REGISTER)
          MEND
```

the address register parameter always refers to a location in memory. Additional examples in which the syntax associated with the parameter determines the meaning of the statement follow:

```
CMP.  \COUNT,(A2)    versus    CMP   #\COUNT,(A2)

MOVE  \COUNT,D4      versus    MOVE  '\COUNT',D4
```

Naming Parameters -- A name preceded by a backslash character (\) represents each parameter within the definition. The names can be any length that fits on the definition's statements and can be composed of any printable ASCII character except the backslash. Examples of legal parameter names follow:

```
\P
\addresses-of-users-table
\NAME
\PRINTER#
\%
\register
```

Each parameter must have a name that is unique within the definition. Different macros, though, can use the same name. Several macros in a module, for instance, might have a parameter named **\register.**

Not only must you ensure that parameter names are unique, you must specify any names that are subsets of other names after the longer name. In this example:

```
EXAMPLE:  MACRO     \OPER,\OPERAND
          \OPER     \OPERAND
          MEND
```

OPER is a subset of **OPERAND**, but is specified before the longer name. When the Assembler expands this macro, it replaces the **OPER** in **OPERAND** with the value for the parameter **OPER.**

This call:

```
EXAMPLE BRA,LOOP
```

produces this nonsense code:

```
          BRA       BRAAND
```

You can get around this problem by specifying the longer name first:

```
EXAMPLE:  MACRO     \OPERAND,\OPER
          \OPER     \OPERAND
          MEND
```

Then when you call the macro:

```
EXAMPLE:  LOOP,BRA
```

you get a usable statement out of the expansion:

```
          BRA LOOP
```

Using Regular Parameters -- Use the macro's regular parameters to substitute instructions or operands or to substitute characters within those parts. The following example shows a macro that uses its parameters to specify values for the operands of a macro:

```
SHIFT:  MACRO       \NO-OF-BITS,\REGISTER
        LSR         #\NO-OF-BITS,\REGISTER
        MEND
```

This call:

```
SHIFT   4,D5
```

produces this code after the Assembler expands the macro:

```
LSR     #4,D5
```

The next example uses the parameter to specify the value of a single character in the macro's instruction.

```
SHIFT:  MACRO       \DIRECTION
        LS\DIRECTION #4,D5
        MEND
```

To right-shift register D5, the macro call is:

```
SHIFT   R
```

which produces the following code:

```
LSR     #4,D5
```

Using the Size Parameter -- Use this parameter to specify the size extension of the operands for one or more instructions. Suppose you want to write a macro that would right-shift bits in long or short words. A macro definition written this way:

```
SHIFT:  MACRO.\SIZE
        LSR.\SIZE               #4,D0
        MEND
```

could be called this way to right-shift a long word:

```
SHIFT.L
```

producing:

```
LSR.L   #4,D0
```

Combining Regular Parameters and Size Extension Parameters -- You can mix the use of size extension parameter with other parameters. This example uses parameters to specify the direction of the shift, the length of the word, and the register for a logical shift instruction:

```
SHIFT:  MACRO.\SIZE             \DIR,\REGIS
        LS\DIR.\SIZE            #4,\REGIS
        MEND
```

This call:

```
SHIFT.L R,D7
```

to this macro produces this code:

```
LSR.L   #4,D7
```

Comments - The Assembler allows two types of comments in a macro definition. A comment that is to appear in the assembly listing is preceded by a single semicolon in the normal fashion. A comment that is not to appear in the assembly listing (that is, a comment that describes the definition only and has no bearing on the source code) is preceded by two semicolons. This saves memory during the assembly and also creates a shorter assembly listing.

EXITM Pseudo Operation Code - When the Assembler encounters the EXITM pseudo operation code, it does not expand the balance of the macro. This can be used to advantage in conjunction with conditional assemblies. In this example:

```
SAMPLE   MACRO   \ARG
           :
           :
         IF      LOCAL
         EXITM
         ENDIF
           :
           :
         MEND
```

If you set **LOCAL** to true, the Assembler executes EXITM and doesn't expand the last portion of the macro. This last portion of the macro might include special processing needed only in modules preparing reports for the local, but not federal, governments.

Argument Substrings - You can access any character or group of characters from a parameter assigned an ASCII value. You reference these characters through this syntax:

```
\parameter-name(first-position,last-position)
\parameter-name(position)
```

where the positions are numeric indices to character positions based on this indexing scheme:

```
 1 = first character
 2 = second character
 .
 .
 .
-2 = second to last character
-1 = last character
```

This scheme allows the same character to be addressed by its position relative to the beginning or end of the string.

The following macro definition provides an example of argument substrings:

```
SAMPLE:  MACRO   \ARG
FLD1:    DC.B    '\ARG(1,26)'
FLD2:    DC.B    '\ARG(1,-1)'
FLD3:    DC.B    '\ARG(1,5)'
FLD4:    DC.B    '\ARG(11,14)'
FLD5:    DC.B    '\ARG(-14,-11)'
FLD6:    DC.B    '\ARG(16)'
FLD7:    DC.B    '\ARG(-7)'
         MEND
```

When called with the alphabet as the argument:

```
SAMPLE   ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

this macro produces:

```
FLD1:    DC.B    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
FLD2:    DC.B    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
FLD3:    DC.B    'ABCDE'
FLD4:    DC.B    'KLMN'
FLD5:    DC.B    'MNOP'
FLD6:    DC.B    'P'
FLD7:    DC.B    'T'
```

Writing the Macro Call

The format of a macro call is:

[label:] name[.size-ext] [param1,param2,...]

where:

label:

is an optional label for the statement.

name

is the name of the macro as given on the MACRO statement of the macro definition.

size-ext

is an optional size extension that the Assembler substitutes for the size extension parameter in the macro definition. The Assembler ignores any size extension in the call if the macro definition does not have a size extension parameter.

A period (.) separates the name and the size extension.

parameter1,parameter2,...

are optional operands that supply values for parameters defined in the macro definition. The values are associated to the parameters in the definition by their order on the call statement. The Assembler assigns the first value to the first parameter, the second value to the second parameter, and so forth. If you do not give a value for a parameter, the Assembler assigns that parameter a null value. If you give more values than the macro has parameters, the Assembler ignores the extra values.

You must separate values with a comma (,).

When calling macros, you must give values for the parameters that are appropriate for their use in the expanded code. For example, if you call this macro:

```
SHIFT:  MACRO          \DIRECTION
        LS\DIRECTION #4,\D5
        MEND
```

using the letter **Q** as the value instead of **R** or **L**:

```
SHIFT   Q
```

causes the Assembler to generate:

```
LSQ     #4,D5
```

Because **LSQ** is not a legal instruction, a syntax error results.

As with any other parameter, the value you give for the size extension parameter must be legal for the way it is used in the macro definition. If the preceding example had a size extension parameter, and you used this call:

```
SHIFT.P  R
```

the Assembler would expand the macro to produce:

```
LSR.P    #4,D0
```

Because **P** is an illegal extension, a syntax error results.

In the same way, giving **B** as the extension for a macro containing this line:

```
CMPA.\SIZE       POINTER,A4
```

produces:

```
CMPA.B   POINTER,A4
```

which also causes a syntax error because you cannot do byte operations on address registers.

Nesting Macros

Macro **definitions** may be nested. A macro definition can contain a macro definition that contains a macro definition, and so on. The Assembler cannot expand a macro definition within a larger, outside macro definition until the larger definition is called. This means that the outside macro should be called before the inside macro to avoid generating an assembly error.

Macro **calls** may be nested to a maximum of eight levels. A macro definition can contain a macro call, whose macro definition contains a macro call, whose macro definition contains a macro call, and so on, up to eight levels. Exceeding this limit generates a nesting error. A macro may also call itself, provided there is a way of ending the self-calling before the ninth level.

The benefit of nesting macro definitions may not be obvious; the following example illustrates one level of nesting used to define several different macros:

```
DEFINE:  MACRO   \1,\2
NG\1\2:  MACRO
         NEG     \1
         NEGX    \2
         MEND
         MEND
```

This nested definition may then be called in a source module as follows:

```
           :
         DEFINE   D0,D1
START:   ...
           :
         NGD0D1
           :
```

The call to DEFINE created the macro **NGD0D1**. Other calls to DEFINE could create macros such as **NGD1D2** or **NGD4D7**. DEFINE must be called once for every macro that it defines--this call must precede the call to the nested macro.

The preceding functions could also be implemented by a single macro:

```
NG:      MACRO   \1,\2
         NEG     \1
         NEGX    \2
         MEND
```

The difference here is that you specify the registers each time you call the macro:

```
           :
         NG      A1,A2
           :
```

CONDITIONAL ASSEMBLY (IF CONDITIONS)

The IF pseudo operation code allows you to write a source module in which certain blocks of code are assembled or not depending on the satisfaction of particular conditions.

This is especially useful in conjunction with the MACRO and *INCLUDE pseudo operation codes. When using the IF pseudo operation code with *INCLUDE, particular files may be included or not depending on values in the source module. Such a file might be a series of macros that are needed in the source module only under certain conditions.

The IF pseudo operation code is useful with macro definitions as a means of determining the desired number of levels of nesting of a macro within itself. The feature may also be used to cause a macro to set up a subroutine the first time the macro is called, and to generate a subroutine call upon subsequent macro calls.

Writing the Basic Conditional Block

The format of the IF pseudo operation code is as follows:

```
(no label)    IF      expression
                :
        source code
                :
(no label)    ENDIF   (no operand)
```

where:

IF

is the IF pseudo operation code.

expression

is any legal expression as defined in Chapter 2. The expression is considered true if it evaluates to any non-zero value.

All terms of the expression must have been previously defined. The expression must evaluate to an absolute quantity.

source code

is the source code to be included in the module if the expression is true.

ENDIF

is an ENDIF pseudo operation code that marks the end of the source code that is to be included if the expression is true.

Writing the Expression

The expression following the IF may be any legal label name, expression, or constant as described in the beginning of this chapter. The Assembler evaluates it to determine whether it is true or false; a false expression is one that evaluates to 0, and a true expression is one that evaluates to -1 (0FFFFh). However, any non-zero value is considered to be true. The IF pseudo operation code evaluates the expression as a 32-bit quantity. All the terms of the expression must have been previously defined to avoid errors; also, the expression must evaluate to an absolute quantity.

An example of an IF pseudo operation code with an expression is:

```
IF      COUNT = 0
```

This generates a value of true (or -1) if COUNT is equal to 0. The example could also be written:

```
COUNT:  EQU     1
          :
          :
        IF      COUNT
```

In this case, COUNT has the value of 1, which also stands for true (non-zero).

Note the difference between the two examples. In the first, COUNT must equal 0 for the expression to be true. In the second, COUNT must equal anything but zero for the expression to be true.

After evaluating the expression, the Assembler assembles the code following the IF pseudo operation code if--and only if--the expression was evaluated to be true. If the expression was false, the block of code bounded by the IF and ENDIF pseudo operation codes is ignored by the Assembler. It is possible to suppress the print listing of such ignored code by using either the NOCOND Option (see Chapter 8) or the LIST NOCOND pseudo operation code (see Chapter 7).

Writing IF-THEN-ELSE Conditional Blocks

The ELSE pseudo operation codes allow conditional assemblies to be written so that one block of code is included if the condition is true and another block of code is written if the condition is false. In this example:

```
        IF      POUND
          :
          :
(code to convert pounds to dollars)
          :
        ELSE
          :
          :
(code to convert francs to dollars)
          :
        ENDIF
```

the code for converting pounds to dollars is included in the module if **POUND** is true and the code for converting francs to dollars is included if **POUND** is false. In an IF-ELSE construction, the code preceding the ELSE is included if the condition is true, and the code following the ELSE is included if the condition is false.

Nesting Conditional Assemblies

IF pseudo operation codes may be nested up to eight levels deep; more than this generates an error message. IF pseudo operation codes may also be nested in macros, making it possible for a macro to call itself the number of times specified by the IF pseudo operation code (an example is given in the following section). Macro parameters may be used in the expression of the IF pseudo operation code. The following example illustrates this:

```
ROTATE:  MACRO   \DIREC
         IF      '\DIREC' = 'R'
         LSR     #4,D0
         ENDIF
         IF      '\DIREC' = 'L'
         LSL     #4,D0
         ENDIF
         MEND
```

Note that the actual ASCII value of the parameter may be specified by enclosing it in single quotation marks as with any ASCII string.

The two IF pseudo operation codes check to see if the parameter specified when calling ROTATE is **R** or **L**. If it is neither, no source code is assembled. If **R** or **L** is specified, the corresponding left or right rotates are generated.

REPEAT EXPANSIONS

The Repeat Expansions feature allows you to write repetitive code in a structured fashion so that it may be more easily written, understood, debugged, and modified. The expansions do not generate code containing loops. Rather, they expand the code as is demonstrated in the examples.

Basic Repeat Expansion

A repeat expansion written with the REPT pseudo operation code repeats the generation of a given section of code a specified number of times. The format is:

```
[label:]  REPT  expression
               :
               :
          source code
               :
               :
             MEND
```

where:

label

is an optional label that is assigned the value of the program counter for the first byte of the first instruction of the first expansion.

REPT

is the REPT pseudo operation code. The label, if any, and the expression appear on the REPT statement.

expression

is any legal expression as defined in Chapter 2 that gives a numeric value defining the number of times the definition is to be repeated.

source code

is the instructions and pseudo operation codes that are to be repeated the number of times specified in the expression. Any operation codes or pseudo operation codes can be used.

MEND

is a MEND pseudo operation code that marks the end of the repeat expansion.

This repeat definition:

```
REPT    256
DC.B    0FFh
MEND
```

generates the following code:

```
        DC.B    0FFh
          :
          :
  (256 times)
          :
          :
        DC.B    0FFh
```

Iterative Repeat Expansion

A repeat expression written with the IRP pseudo operation code repeats a given section of code, substituting a new value for a given argument, until it runs out of values.

The format is:

```
[label:]  IRP  \arg,value1[,value2,value3,...]
                :
                :
          source code
                :
                :
             MEND
```

where:

label

is an optional label that is assigned the value of the program counter for the first byte of the first instruction of the first expansion.

\arg

is the argument that is substituted with the values given in this statement. The argument name can be any valid label preceded by a backslash character (\).

value1,value2,value3,...

are the values to be substituted for the argument in the expansions. The values can be any quantity or expression that is appropriate for use in the expansion. The values are substituted exactly as they appear in the IRP statement. At least one value must be given; as many more can be given as will fit on one statement. The values must be separated by a comma. The Assembler ignores SPACEs and TABs except within values.

source code

is the instructions and pseudo operation codes that are to be repeated the number of times specified in the expression. Any operation codes or pseudo operation codes can be used.

MEND

is a MEND pseudo operation code that marks the end of the repeat expansion.

This iterative repeat definition:

```
IRP     \VAR,X,Y,Z
MOVE    D4,\VAR
MEND
```

generates the following code:

```
MOVE    D4,X
MOVE    D4,Y
MOVE    D4,Z
```

Iterative Repeat Expansion with Characters

A repeat expression written with the IRPC pseudo operation code repeats a given section of code, substituting a new character for a given argument, until it runs out of characters. The format is:

```
[label:]  IRPC  \arg,'character-string'
                :
                :
          source code
                :
                :
             MEND
```

where:

label

is an optional label that is assigned the value of the program counter for the first byte of the first instruction of the first expansion.

\arg

is the argument that is substituted with the values given in this statement. The argument name can be any valid label preceded by a backslash character (\).

'character-string'

is a string of characters, each of which is used once as the substitution for the argument. Any valid ASCII character can be used within the string. The string must be enclosed by single quotation marks ('). At least one character must be given; as many more can be given as will fit in one instruction.

source code

is the instructions and pseudo operation codes that are to be repeated the number of times specified in the expression. Any operation codes or pseudo operation codes can be used.

MEND

is a MEND pseudo operation code that marks the end of the repeat expansion.

This iterative repeat definition:

```
IRPC     \CHAR,'STRING'
MOVE.B   #\CHAR,(A0)+
MEND
```

generates the following code:

```
MOVE.B   #'S',(A0)+
MOVE.B   #'T',(A0)+
MOVE.B   #'R',(A0)+
MOVE.B   #'I',(A0)+
MOVE.B   #'N',(A0)+
MOVE.B   #'G',(A0)+
```

Chapter 4

PREPARING MODULES FOR LINKING

Assembling your program is just one step in preparing it for execution. Linking is just as important. This chapter describes the features the Assembler provides to control that linking.

Before writing an assembly program, you should become familiar with the capabilities of the 68000 Linker because its capabilities may affect the amount of linking information you put in the source code of a module.

BASIC REQUIREMENTS FOR LINKING MODULES

At the minimum, the linker needs a name for each file containing a module it is to link together to form a program, and it needs a starting address for the program.

Module Names

The linker needs a name that uniquely identifies each module to be linked together into a program. The name of a module can be assigned with a NAME pseudo operation code placed within the module. If a name is not assigned this way, the linker uses the first 1-32 characters of the module's filename as the name. If the filename is 32 characters or less, the linker uses the entire filename as the linker name.

Defining the Starting Address

In any program, but especially in multi-module programs, you often do not want execution to start at the beginning of the linked program either because the program begins with a data area or because it does not begin with the correct program area. If you specify a location at which execution is to begin, the linker

ensures that execution begins at that location. In this way, any intervening data areas or program code are jumped over.

You can specify a program's starting address within the program with the END pseudo operation code. While the main purpose of END is to mark the end of a module, in one module of a program it can also be used to specify a starting address for the program. The following example shows the easiest way to do this:

```
START:   LEA      ...
           :
           :
           :
         END      START
```

In this example, the label **START** marks the starting address. The END statement then uses the START label to specify that execution of the program is to begin at START.

The starting address specified with END must be within the module itself. Also, if you specify starting addresses in two or more modules of the same program, the linker uses the first address specified and ignores the subsequent address.

RESOLVING GLOBAL LABELS

The Assembler and the linkers recognize two kinds of labels in a module: those that are local to a module and those that are globally available to all modules in a program. With one exception, you define and use the two types identically. The difference is that you can use global labels in modules other than the one in which you assign their value. Because of this, it is often the linker that must resolve the value for global labels--not the Assembler as is the case with local labels.

For the linker to resolve the values of global labels, you must:

-- Use the ENTRY pseudo operation code to declare in each module those labels that are referenced in other modules. The ENTRY statement informs the linker that this module defines the named labels.

-- Use the EXTERN pseudo operation code to declare in each module those labels whose values are declared in other modules. The EXTERN statement prevents the Assembler from flagging the named labels as undefined.

The following example comes from a module that uses ENTRY to declare that the values for three labels are defined in that module. The module contains the code for two subroutines and a conversion table used to convert metric measurements to English measurements, and vice versa.

```
            ENTRY   METRIC,ENGLISH,CONVERSION_TABLE

REM         METRIC TO ENGLISH CONVERSIONS
METRIC:     ...
              :
            RTS

REM         ENGLISH TO METRIC CONVERSIONS
ENGLISH:    ...
            :
            RTS
REM         CONVERSION TABLES
CONVERSION_TABLE:
            ...
              :
            END
```

The next example is from another module that uses EXTERN to declare that the values for three labels are defined in another module. This program calls both subroutines used in the previous example and uses the conversion table to pass parameters.

```
            EXTERN  METRIC,ENGLISH,CONVERSION_TABLE
START:      ...
              :
            LEA     CONVERSION_TABLE,A0
            MOVE    (A3),D3
            JSR     METRIC
              :
            ADDQ    #1,A0
            MOVE    (A0),D3
            JSR     ENGLISH
            END     START
```

Just as you cannot define local labels more than once within a module, you cannot define global labels more than once within a program. If you do multiply define a global label, it is impossible to determine what its value will be at any given time.

If a module lists a label on an ENTRY statement and no other module lists the label on an EXTERN statement, the linker treats the label as a local label. On the other hand, if a module lists a label on an EXTERN statement and no other module lists the label on an ENTRY statement, the linker treats the label as an undefined label and gives an error message.

PROGRAM SEGMENTS

A program segment is a block of code that shares common attributes, such as the addressing mode, and that the linker treats as a unit within the program. Program segments are similar to the code segments of some Assemblers. An example may make the concept of program segments and their use clearer. Assume you are writing a program that will eventually have the program portions placed in Read Only Memory (ROM) and the data portions placed in Random Access Memory (RAM). You might write the source code such that the data areas are defined immediately after the routines using them.

By using program segments, you can specify the type of addressing for each program segment (the program area might be absolute and the data area might be relocatable) and have the linker separate the two areas.

There are two types of program segments: standard segments the Assembler provides and user-defined segments.

The standard segment is the Blank segment (denoted by empty string ''), intended for relocatable program code. This segment meets the requirements of most programs. When you need additional segments or segments with special attributes, you can define additional segments.

You use the PSECT pseudo operation code to define program segments. The code that follows each PSECT statement with the same name is part of the same program segment. The remainder of this section describes the possible attributes for a program segment.

Options for Program Segment Attributes

REA If a program segment has the REA attribute, it is a readable segment. If it does not have this attribute, the processor is not supposed to read the data from the program segment.

WRI If a program segment has the WRI attribute, it is a writable segment, i.e., the processor should be allowed to write into the segment. If a program segment does not have this attribute, the processor is not supposed to write into it.

EXE A program segment that has the EXE attribute is an executable segment. You may branch (jump) only to an address in an executable segment. If a program segment does not have this attribute, the processor cannot branch into it.

SHA A program segment that has this attribute is sharable in the sense that more than one process has access to it. If it does not have this attribute, the program segment is intended to be used by only one process.

COM A program segment with this attribute is a common program segment that can be used to implement FORTRAN-like common blocks. Suppose that two modules declare a program segment named ABC, with the REA and WRI attributes. The linker will concatenate them into a larger unit. The linker would concatenate them even if they had different names, as long as they have identical attributes. However, if the segment ABC also has the COM attribute, the linker will not concatenate the ABC segments from both modules, but will "overlay" them in the sense that both modules will access the same locations in the memory when they use the ABC segment. In the case of common segments, the segment name is important.

ABS A segment with this attribute is not supposed to be relocatable. At present, the linker does not support absolute segments.

THE STANDARD PROGRAM SEGMENT

Every program contains the default program segment, even if its length is zero. The default program segment has no name (or has the name '') with the attributes REA, WRI, EXE. For the majority of programs, this is all you need. Different program segments can be introduced by the **PSECT** pseudo operation code (see Chapter 7).

Chapter 5

DECLARING VALUES AND RESERVING MEMORY

The Assembler provides several pseudo operation codes that you can use to declare values and to reserve memory for data areas. This chapter summarizes the functions of these pseudo operation codes so that you can select those that meet your needs. Before using any of these pseudo operation codes, you should read its complete description in Chapter 7.

RESERVING MEMORY

The Assembler provides two pseudo operation codes for reserving memory that is to be used for data areas. These are:

DS{.B,.W,.L} The DS pseudo operation code reserves a specified number of bytes, 2-byte words, or 4-byte words of memory. The Assembler does not initialize the bytes.

In this example:

```
ADDRSTABL:   DS    20
```

the Assembler would reserve 20, 2-byte words of memory that might be used as a table of 20 entries (2 bytes per entry).

DC{.B,.W,.L} The DC pseudo operation code reserves a string of bytes, 2-byte words, or 4-byte words and initializes the string with the value or values given in the statement. As much memory is reserved as is needed to store the value(s).

In this example:

```
LOGIN:  DC.B  'TERMINAL LOGGED IN\n\0'
```

the quotes will cause the string to be converted to ASCII characters and stored in consecutive bytes.

In this example:

```
DC   -2,-4,-6,10,11,17
```

the numbers are converted to binary and stored in consecutive 2-byte words.

When reserving bytes, the field may end on an odd address. If the subsequent information is also built from bytes, nothing is wrong. However if the information contains words, lengths, or instructions (which must reside on an even address), the Assembler forces the address to the next even value and issues a warning. You can avoid these warning messages if you always reserve an even number of bytes, or, more simply, by following each DC.B or DS.B statement by an ALIGN pseudo operation:

```
LOGIN:   DC.B    'LOGIN\n\0'
NAME:    DS.B    23
         ALIGN
```

Declaring zero (0) values for DS and DC has different effects for the two pseudo operation codes as described here:

DS 0 Specifying zero with DS has the Assembler not reserve any memory.

DC 0 Specifying zero with DB has the Assembler reserve one 2-byte word of memory initialized with the value zero.

EQUATING VALUES

The Assembler provides three basic ways to equate a label with a value that can be any legal constant, address, or expression:

EQU The EQU pseudo operation code allows you to equate a label to a value. A label assigned a value with EQU cannot have that value re-defined later on in a module.

SET The SET pseudo operation code allows you to equate a label to a value. SET differs from EQU in that values declared with SET can be re-defined within a module.

STRUCT The STRUCT pseudo operation code equates values that are used to define a series of offsets from a base. STRUCT is described in detail later in this section.

Equated values are local to the module in which they are defined and cannot be used as global labels (see Chapter 4).

The EQUATE pseudo operation code is useful for simplifying or clarifying source code. For example, suppose the ASCII characters for carriage return (CR) and line feed (LF) were to be used throughout a source program. Instead of using their values, a clearer procedure would be to include the statements:

```
CR      EQU     0Dh
LF      EQU     0Ah
```

somewhere in the source program and then use the labels **CR** and **LF** to stand for the values as in:

```
STRING:   DB   'end of text',CR,LF
```

You can also use EQUATE to quickly change the value of a quantity throughout a program. Suppose that you are testing a program with different values for a timer. Suppose further that this value is used 10 times throughout the source code. If the original value is used in each of those 10 places, then you must change the timer value in all 10 places to test a new time increment. However, if each of the 10 places uses the label **TIMER** and the following statement appears somewhere in the module:

```
TIMER     EQU       value
```

then the **TIMER** value can easily be changed. This assures upon re-assembly that all the places **TIMER** is used will be changed.

The STRUCT pseudo operation code is defined differently than any of the other equate pseudo operation codes. It is used to define a series of equated values that are usually used to reference tables. The format of the STRUCT construction is:

```
            STRUCT          expression
[label]     DS{.B,.W,.L}    size
[label]     DS{.B,.W,.L}    size
              :
              :
              :
            MEND
```

where the expression on the STRUCT statement gives the initial offset, the labels on the DS statements are the labels to be equated, the size on the DS statements give the offset values, and the MEND statement ends the construction. The DS statements used in a STRUCT construction **do not** reserve memory; they merely specify offsets. The following structure:

```
            STRUCT  10
FIELD1      DS      4
FIELD2      DS      2
FIELD3      DS      4
            MEND
```

is the equivalent of:

```
FIELD1      EQU     10
FIELD2      EQU     18
FIELD3      EQU     22
```

The advantage of STRUCT is that it automatically calculates the offsets, making it easy to add fields, re-arrange their order, or change their lengths without having to specify new values for each of the other fields.

You might use STRUCT when using several tables that have identical formats. You might, for example, have three tables of 100 bytes each that have identical formats. You could address them by assigning a label to each field of each table as is done here for one field:

```
TBL1FLD4 DS      4
```

Then, however, you must code those labels into your program as follows:

```
LEA      A1,TBL1FLD4
```

Referencing fields this way reduces the flexibility of your code because your programs must use the labels for each field.

With STRUCT, you can write general routines that use offsets to reference the fields. You could define offsets that apply to several tables, for example:

```
         STRUCT  0
FIELD1   DS      4
FIELD2   DS      2
FIELD3   DS      4
         MEND
           :
           :
         LEA     TABLE1,A2
         BSR     VERIFY
         LEA     TABLE2,A2
         BSR     VERIFY
           :
           :
VERIFY   ...
         MOVE    FIELD2(A2),D0
           :
           :
```

In this case, the fields of **TABLE1** and **TABLE2** can be referenced by loading the starting address of the tables in a register and then adding the offsets to the starting address. This approach gives you flexibility because you can easily change the structure of the tables without having to rewrite other sections of code. It also allows you to write general routines, such as the **VERIFY** subroutine of this example, that can work with either table.

A common practice with STRUCT is to use the value of the offsets to define the lengths of the tables to which the offsets apply as in this example:

```
          STRUCT  10
FIELD1    DS      4
FIELD2    DS      2
FIELD3    DS      4
LENGTH    DS      0
          MEND
            :
            :
TABLE1    DS      LENGTH
TABLE2    DS      LENGTH
```

The advantage of using the offset value for **LENGTH** to define the lengths of the tables is that if you change the structure of the offsets in any way, the length of the tables is automatically changed, too.

Chapter 6

THE ASSEMBLY LISTING

The Assembler produces an assembly listing for each program assembled that can include a program listing and a symbol table. This chapter describes the listing and its options. The last section of this chapter contains a sample listing.

The Assembler normally places the assembly listing in a file of the name:

filename.prn

where the filename is the same name as the filename of the file containing the source code and extension changed to **prn.** This source file normally resides in the current directory, or in the directory specified by -asm option (see Chapter 8). By means of the -prn option (Chapter 8), the listing file may be placed in any directory provided that the user has the correct access privileges to write into that directory.

PAGE LAYOUT

The Assembler allows you considerable control over the page layout of the listing. The only fixed feature of the listing is the page heading.

The following features of the page layout can be changed:

Page Length

The default page length of 59 lines can be changed with the PAGE option of the Assembler call. (Refer to Chapter 8.)

Page Width

The default page width of 128 characters can be changed with the TRUNC option of the Assembler call. (Refer to Chapter 8.) Any lines longer than selected length are truncated. The WIDTH option of the Assembler Call can be used to wrap lines longer than the specified width to the next line. The TRUNC and WIDTH options are mutually exclusive.

Title

A title, which will appear one line below the page heading, can be specified with the TITLE pseudo operation code. (Refer to Chapter 7.) If you do not specify a title, the Assembler will use the filename of the source code file.

Subtitle

A subtitle, which will appear two lines below the page heading, can be specified with the SUBTTL pseudo operation code. (Refer to Chapter 7.) If you do not specify a subtitle, none will appear in the listing.

ADDRESS SYMBOLS USED IN ASSEMBLY LISTING

Three symbols are used to flag addresses in the program listing and cross-references to indicate the type of code (for example, relocatable) to which they belong. These addresses are flagged wherever they occur in the program. For example, this object code listing of an instruction:

```
000000 33FA 0006  0000000A'        0001            move    a,b
                                   0002  ;
000008 (00000002)                  0003  a:        ds      1
00000A (00000002)                  0004  b:        ds      1
```

Symbol 0000000A, in the first line, is flagged with a single quotation mark because it is the address relative to the beginning of the current segment of code.

The following symbols are used to mark the addresses that will be modified by the linker.

Single Quotation Mark (')

The Assembler marks all addresses that are relative to the beginning of the current PSECT by a single quotation.

Percent (%)

The Assembler marks all addresses that are relative to the beginning of some other PSECT by a percent sign.

Pound Sign (#)

The Assembler marks all addresses that are relative to some external symbol by a pound sign.

PROGRAM LISTING

The program listing contains a listing of the original source code, the assembled object code, and error messages, if any.

Each of the items in the program is described below:

Address

The address is the hexadecimal address of the first byte of memory that is created by the instruction being listed on this line. No address is given for those pseudo operation codes that do not require memory.

Object Code

The instruction or data created by the operation code or pseudo operation code being listed on this line is displayed in this column. No object code is given for those pseudo operation codes that do not require memory or for lines that only have a comment on them.

Symbols Used to Mark Source Code Statements

The Assembler uses four symbols to flag source code statements in the program listing:

Asterisk (*) - The Assembler places an asterisk (*) before the line number of an instruction that uses a longer address form than necessary.

P - The Assembler marks all privileged instructions by a letter P immediately before the line number.

Plus Sign (+) - The Assembler places a plus sign after the line number of any statement that is included in the program listing as the result of a macro expansion.

Minus Sign (-) - If the STRUCT pseudo operation code is used to equate offset values, the statements used to define the offsets are marked with a minus sign after the line number.

Statement Line Number - This column gives the ordinal number of the source code line in the program. Each line of the source code is numbered whether it is listed or not. (Several options of the LIST pseudo operation code [refer to Chapter 7] and the Assembler call statement [refer to Chapter 8] control whether or not certain parts of the source code are listed.)

Source Code Statement - The original line of source code is listed in the last column exactly as it appears in the source code. Portions of each source code statement may be truncated (see the TRUNC option in Chapter 8) or wrapped around to the next line (see the WIDTH option in Chapter 8).

ERROR MESSAGES

The Assembler flags all source code lines that contain errors with one of the error messages listed in Chapter 9.

OPTIONS

The Assembler gives you a number of options to control what information is printed in the program listing.

Turning Off the Program Listing

The printing of the program listing can be controlled with the LISTON and LISTOFF options of the Assembler call statement (refer to Chapter 8) or with the ON and OFF options of the LIST pseudo operation code (refer to Chapter 7). The Assembler call options control the printing of the entire program listing while the LIST options control only the printing of the program listing following their appearance in the source code.

The default is to print the entire program listing.

Listing False Conditional Assembly Blocks

The printing of the source code that is part of false conditional assembly blocks (also known as IF assembly blocks) is controlled with the COND and NOCOND options of the Assembler call statement (refer to Chapter 8) and with the COND and NOCOND options of the LIST pseudo operation code (refer to Chapter 7). The default is to list the source code of the conditional assembly blocks found to be false during the assembly.

The source code of true conditional assembly blocks is always listed unless the program listing is suppressed with the LISTOFF option of the Assembler call statement.

The Assembler call options control the printing of the entire program listing while the LIST options control only the printing of the program listing following their appearance in the source code.

Listing the Object Code Created by DC Pseudo Operation Codes

The printing of the object code that is generated by the DC pseudo operation code is controlled with the TEXT and NOTEXT options of the Assembler call statement (refer to Chapter 8) and with the TEXT and NOTEXT options of the LIST pseudo operation code (refer to Chapter 7). The default is to list only one line for each DC instruction.

The DC pseudo operation code itself is always listed unless the program listing is suppressed with the LISTOFF option of the Assembler call statement. Complicated DC instructions may produce more than one line of listing unless the NOTEXT option is in effect.

The Assembler call options control the printing of the entire program listing while the LIST options control only the printing of the program listing following their appearance in the source code.

Listing Macro Call Expansions

The printing of the source code that is generated by the expansion of macro calls is controlled with the GEN and NOGEN options of the Assembler call statement (refer to Chapter 8) and with the GEN and NOGEN options of the LIST pseudo operation code (refer to Chapter 7). The default is to list the source code created by the expansion of macro calls.

The macro call itself is always listed unless the program listing is suppressed with the LISTOFF option of the Assembler call statement.

The Assembler call options control the printing of the entire program listing while the LIST options control only the printing of the program listing following their appearance in the source code.

Formfeeds

The FORM and EJECT pseudo operation codes (refer to Chapter 7) cause the listing to advance to the top of the next page whenever they are found in the source code. The pseudo operation codes themselves are never listed.

Cross-Reference Table

The Assembler can currently print a symbol table only.

Symbol Table

The symbol table contains an alphabetical listing of all symbols used in the program and their values. When the value is an address that is not absolute, the address is followed by one of the address symbols described earlier in this chapter. The value given for symbols defined by EXT pseudo operation codes is the address of their first use in the program.

The SYMBOL option of the Assembler call statement (refer to Chapter 8) must be specified to get a copy of this table. Any symbols following the NOXREF option of the LIST pseudo operation code but preceding an XREF option of LIST or the end of the source code will not be included in the symbol table.

Chapter 7

PSEUDO OPERATION CODES

This chapter describes each of the pseudo operation codes recognized by the Assembler and gives examples of their use. The description of the pseudo operation code gives a reference to the chapter describing its use.

ALIGN--ALIGN DATA FIELDS

Use the ALIGN pseudo operation code to force the location counter on a word (long word) boundary.

The format is:

```
ALIGN{.w,.l}
```

CONMSG--CONSOLE MESSAGE

Use the CONMSG pseudo operation code to have the Assembler send a message to the console during the second pass of the Assembler.

The format is:

```
CONMSG [any-message]
```

where:

any-message

is any string of ASCII characters. The string can be any length and is not enclosed in quotes.

In the following example:

```
        :
        :
IF      MONTH EQ 12
CONMSG  YEAR-END CALCULATIONS INCLUDED
        :
        :
ENDIF
        :
        :
```

the message, "YEAR-END CALCULATIONS INCLUDED" is sent to the console if the conditional block is included in the object code.

DC--DEFINE CONSTANT

Use the DC pseudo operation code to reserve a block of memory as a data area initialized with the values given in the statement. As much memory will be reserved as is needed to hold the values given.

The format is:

[label:] DC{.B,.W,.L} values

where:

label:

is an optional label that the Assembler will assign the value of the program counter for the first byte of the data area.

.B

has the Assembler reserve one byte for each value given in the statement.

.W

has the Assembler reserve one 2-byte word for each value given in the statement. This is the default.

.L

has the Assembler reserve one 4-byte word for each value given in the statement.

values

are values that may be constants, as described in Chapter 2, ASCII strings enclosed in single quotes ('), or expressions, as described in Chapter 2. As many values can be given in a single statement as will fit on that statement.

ASCII strings are treated as a sequence of quoted characters, so that

```
DC.B    'abc'
```

is equivalent to

```
DC.B    'a', 'b', 'c'
```

Note that the statement

```
DC      'ab'
```

produces the same result as

```
DC      'a', 'b'
```

i.e., two words, containing 0061h and 0062h, respectively.

Chapter 5 gives examples of the use of DC.

DROP

The format is:

```
[label:]   DROP   <A-register>
```

This pseudo instruction informs the Assembler that the specified address register <A register> is no longer to be used for addressing purposes by the Assembler. See also the USING pseudo instruction.

DS--DEFINE STORAGE

Use the DS pseudo operation code to reserve a block of memory that is not initialized with any value.

The format is:

[label:] DC{.B,.W,.L} expression

where:

label:

is an optional label that the Assembler will assign the value of the program counter for the first byte of the data area.

.B

has the Assembler reserve the number of bytes specified by the expression.

.W

has the Assembler reserve the number of 2-byte words specified by the expression. This is the default.

.L

has the Assembler reserve the number of 4-byte words specified by the expression.

expression

is a legal expression as defined in Chapter 2 that specifies the amount of memory to be reserved.

Chapter 5 gives examples of the use of DS.

When used with the STRUCT pseudo operation code, DS equates values to be used as offsets rather than reserving memory. For more information on this use of DS, see Chapter 5.

EJECT--PAPER EJECT

Use the EJECT pseudo operation code to have the Assembler advance the assembly listing to the top of the next page.

The format is:

(no label) EJECT (no operands)

EJECT is used for clarity in a print listing. For example, the beginning of a routine can be more clearly identified if it starts at the top of a page.

The EJECT pseudo operation code in the source code will not be printed on the listing. Multiple EJECT pseudo operation codes are ignored. The FORM pseudo operation code may be used in exactly the same way as EJECT to force a paper feed to the top of the next page.

ELS--CONDITIONAL EXECUTION

The ELS pseudo instruction code marks the end of conditional code initiated by an IFcc instruction. Simultaneously it means the start of code that is to be executed if the condition stated in the IFcc instruction is not true (see the IFcc pseudo operation code).

The syntax is:

(no label) ELS[.S,.W]

As the ELS pseudo instruction is converted to a branch instruction, the size extension supplied is applied to the branch instruction.

ELSE--CONDITIONAL ASSEMBLY

Use the ELSE pseudo operation code to mark the beginning of a block of code in a conditional assembly block that the Assembler is to include if the condition tested in the IF statement is false.

The format is:

(no label) ELSE (no operand)

Chapter 3 shows the use of ELSE in conditional assemblies.

END--END OF ASSEMBLY

Use the END pseudo operation code to terminate the assembly of a module and to specify the address where execution of the program is to begin (transfer address).

The format is:

```
[label:]  END  [expression]
```

where:

label

is a label that will be assigned the current value of the program counter.

expression

is a value or label that specifies the starting address of the program. Use the expression only in the module in which execution of the program is to begin. The expression can be any legal expression as defined in Chapter 2 or any label defined in that module.

Chapter 4 describes the use of the END pseudo operation code to specify the address at which program execution is to begin.

Following is a sample use of the END pseudo operation code to terminate assembly of a main module:

```
          ENTRY   MAIN
MAIN:     LEA     A7,1800h
            :
            :
          END     MAIN
```

This sample shows termination of a sub-module to be linked to the main module:

```
BEGIN:   MOVE    #10,D2
           :
           :
         END
```

The END pseudo operation code is a signal to the Assembler that a logical body of code is complete. Therefore, only one END pseudo operation code should appear in a module. Should the END appear in the middle of a block of code, everything following the pseudo operation code will be ignored by the Assembler.

ENDIF--END OF CONDITIONAL ASSEMBLY

Use the ENDIF pseudo operation code to mark the end of a conditional assembly block.

The format is:

(no label) ENDIF (no operand)

Chapter 3 shows the use of ENDIF pseudo operation codes in conditional assemblies.

ENTRY--ENTRY LABELS

Use the ENTRY pseudo operation code to declare that a module contains definitions for the global labels listed. The labels must have their values assigned within the module.

The format is:

(no label) ENTRY label1[,label2,...]

where:

label1,label2,...

are the names of global labels defined within the module.

As many labels may be named in a single statement as will fit. Multiple ENTRY statements can be given in a module.

ENTRY statements may appear anywhere within a program module, but are typically written at the top of a file to be easily found in the print listing.

EQU--EQUATE

Use the EQU pseudo operation code to assign a value to a label.

The format is:

label EQU expression

where:

label

is the label to be equated with the value.

expression

is any legal expression, as defined in Chapter 2, specifying the value. All the terms of the expression must have been previously defined.

Chapter 5 gives examples of the use of equated values.

EXITM--END MACRO EXPANSION

Use the EXITM pseudo operation code to unconditionally halt the expansion of a macro. The EXITM pseudo operation code unconditionally halts the expansion of a macro expansion.

The format is:

(no label) EXITM (no operand)

Chapter 3 gives examples of the use of EXITM statements.

EXTERN--EXTERNAL LABELS

Use the EXTERN pseudo operation code to declare that global labels used in a module are assigned values in another module of the program.

The format is:

(no label) EXTERN label1[,label2,...]

where:

label1,label2,...

are the names of global labels defined in other modules.

As many labels may be named in a single statement as will fit. Multiple EXTERN statements can be given in a module.

EXTERN statements may appear anywhere within a program module, but are typically written at the top of a file to be easily found in the print listing.

Note that a label name declared as an external to a module may not be redefined (i.e., used in the label field) within that module.

FI--IFcc THEN ELS TERMINATING SYMBOL

The FI pseudo instruction marks the end of the conditional block of code initiated by an IFcc or the ELS instruction (see the IFcc pseudo instruction code).

The format is:

(no label) FI

FORM--PAPER FORMFEED

Use the FORM pseudo operation code to have the Assembler advance the assembly listing to the top of the next page.

The format is:

(no label) FORM (no operands)

FORM is used for clarity in a print listing. For example, the beginning of a routine can be more clearly identified if it starts at the top of a page. The FORM pseudo operation code in the source code will not be

printed on the listing. Multiple FORM pseudo operation codes are ignored. The EJECT pseudo operation code may be used in exactly the same way as FORM to force a paper feed to the top of the next page.

IF--BEGIN CONDITIONAL ASSEMBLY

Use the IF pseudo operation code to define a conditional expression that the Assembler will evaluate to decide whether or not to include the block of code following the IF statement. If the expression is true, the Assembler includes the block; if it is false, the block is not included.

The format is:

(no label) IF expression

where:

expression

is any legal expression, as defined in Chapter 2. The expression is considered true if it evaluates to any non-zero value.

All the terms of the expression must have been previously defined. The expression must evaluate to an absolute quantity.

The IF pseudo operation code is used with the ELSE and ENDIF pseudo operation codes to define conditional assembly blocks. See Chapter 3 for more information on these blocks.

IFcc--IF THEN ELS PROGRAMMING STRUCTURE

The syntax of IFcc pseudo instruction is:

```
[label:]    IFcc[.S,.W]
               ...
               ELS
               ...
               FI
```

Size extension, if used, is applied to the branch instruction to which IFcc is converted.

To simplify writing conditional statements in the assembly language, the Assembler has special pseudo operation codes. For example, the program fragment

```
        BNE   LABEL
        <some instructions>

label:
```

has the meaning that <some instructions> are executed if the zero flag in the CCR is set as the result of some previous instruction. The same program segment can be written in a more structured way:

```
IFEQ
<some instructions>
FI
```

There are 14 such conditional instructions:

```
IFCC, IFCS, IFEQ, IFGE, IFGT, IFHI ,IFLE

IFLO, IFLT, IFMI, IFNE, IFPL, IFVC ,IFVS
```

one for each possible branch condition. Such instructions are referred to by the common name IFcc. Each instruction is translated into conditional branch on the complementary condition to a location defined by the FI statement. The lines of code between the IFcc and the FI instruction form the conditional block of code. There is also an extended form of the conditional structure, for example:

```
CMP    #5,d0
IFGT
<some code>
ELS
<some other code>
FI
```

This program fragment is equivalent to

```
            CMP     #5,D0
            BLE     LABEL1
            <some code>
            BRA     LABEL2
LABEL1:     <some other code>
LABEL2:
```

with the obvious meaning that if the value in the D0 register is greater than 5, then <some code> is executed, else <some other code> is executed.

The IFcc instructions offers two benefits:

- You do not have to invent labels to mark the position you want to jump to.
- Programming is more natural. Instead of writing "if condition is not true, then jump over some code," you really write "if condition is true, then execute some code."

Conditional structures formed by IFcc may be nested to 16 levels, i.e., there may be an IFcc structure within an IFcc structure, and so on, up to 16 times.

*INCLUDE--INCLUDE SOURCE CODE FILE

Use the *INCLUDE pseudo operation code to specify a file containing assembly language source code that the Assembler is to insert within the module it is assembling.

The format is:

```
*INCLUDE    'pathname'
```

where:

pathname

is the name of the file to be included.

The *INCLUDE pseudo operation code must begin with the asterisk in column 1. No label field is permitted with this operation code. The quoted pathname may follow the *INCLUDE after at least one delimiter (SPACE or TAB).

All of the named file is included in the present file. If the included file has an END pseudo operation code, this END pseudo operation code will terminate assembly of the including module when it is encountered.

An example illustrates this. Suppose this is the source file to be assembled:

```
BEGIN:    LEA     D7,...
            :
            :
*INCLUDE '/USERS/ACCT/USERFILE.ASM'
          LEA     A0,...
          BSR     ...
          END
```

and suppose the following is USERFILE.ASM:

```
START:    MOVE    D1,...
          MOVE    D2,...
          END
```

Because the USERFILE contains an END pseudo operation code, the Assembler will never see the second LEA and the BSR statements. Assembly will be terminated following the inclusion of USERFILE. To avoid this problem, simply leave off the END pseudo operation codes of files that are to be included in the assembly of other files, or put the *INCLUDE pseudo operation code as the last one in a source program and leave off that source's END pseudo operation code.

The *INCLUDE pseudo operation code is particularly useful in conjunction with conditional assembly blocks of code. (See the discussion of conditional assembly in Chapter 3.) For example, a file may be included depending on whether or not an IF pseudo operation code is satisfied. Also, the IF pseudo operation code can be used to determine which of several files will be included.

An example of this use of *INCLUDE follows. One of three different files will be included and the others ignored depending on the value of the label DECIDE (defined earlier in the source):

```
          :
          :
        IF      DECIDE EQ 0
*INCLUDE 'MOVROUTN.ASM'
        ENDIF
        IF      DECIDE EQ 1
*INCLUDE 'SAVROUTN.ASM'
        ENDIF
        IF      DECIDE EQ 2
*INCLUDE 'LOADROUT.ASM'
        ENDIF
          :
          :
```

*INCLUDEs may be nested up to eight levels; more than this will generate a nesting error.

IRP--ITERATIVE REPEAT

Use the IRP pseudo operation code to write a repeat expansion definition that substitutes a new value for the argument each time the definition is repeated. The expansion is repeated until each argument has been used once.

The format is:

```
[label:]  IRP  \arg,value1[,value2,value3,...]
```

where:

label

is an optional label that will be assigned the value of the program counter for the first byte of the first instruction of the first expansion.

\arg

is the argument that will be substituted with the values given in this statement. The argument name can be any valid label preceded by a backslash character (\).

value1,value2,value3,...

are the values to be substituted for the argument in the expansions. The values can be any quantity or expression that is appropriate for its use in the expansion. The values are substituted exactly

as they appear in the IRP statement. At least one value must be given; as many more can be given as will fit on one statement. The values must be separated by a comma.

Chapter 3 contains a complete description of repeat expansions.

IRPC--ITERATIVE REPEAT WITH CHARACTERS

Use the IRPC pseudo operation code to write a repeat expansion definition that substitutes a new character from a string of characters for the argument each time the definition is repeated. The expansion is repeated until each character has been used once.

The format is:

```
[label:]  IRPC  \arg,'character-string'
```

where:

label

is an optional label that will be assigned the value of the program counter for the first byte of the first instruction of the first expansion.

\arg

is the argument that will be substituted with the values given in this statement. The argument name can be any valid label preceded by a backslash character (\).

'character-string'

is a string of characters, each of which will be used once as the substitution for the argument. Any valid ASCII character can be used within the string. The string must be enclosed by single quote marks (').

Chapter 3 contains a complete description of repeat expansions.

JSYS--CROMIX SYSTEM CALLS

JSYS is a pseudo operation code which is translated into TRAP 0 instruction, followed by the system call number. The syntax is:

[label:] JSYS <system call number>

where:

system call number

is an immediate operand whose value is the system call number.

System call numbers are defined in the file **/equ/jsysequ.asm.** To use a system call, load the registers with required values and execute the appropriate JSYS instruction. As in Z80 Cromix, if an error is found during the system call, the carry flag is set in the CCR, and the register D0 contains the error number. The program HELLO, given at the beginning of chapter 2, is an example of how to use JSYS.

LIST--ASSEMBLY LISTING OPTIONS

The LIST pseudo operation code is used to set the Assembler print-listing options. The options set do not affect the actual object code produced by the Assembler. They simply suppress undesired or repetitive sections of the assembly listing. The format of the LIST pseudo operation code is:

(no label) LIST [option1,option2,...]

where:

option1,option2,...

are the LIST options described in the following paragraphs. The number of options that may be placed on a line is limited only by the line length. However, five is the practical limit because more than this will result in duplicate or conflicting options. Options may be given in any order.

If conflicting options are given (conflicting options are the pairs GEN-NOGEN, COND-NOCOND, ON-OFF TEXT-NOTEXT, and XREF-NOXREF), only the last one of the pair on the line will be used.

The LIST pseudo operation code may be used as often as desired throughout a source code file. However, the following options of the Assembler call override any specifications made with the LIST pseudo operation code

-cond	-nocond
-gen	-nogen
-liston	-listoff
-text	-notext
-xref	-noxref

(Refer to Chapter 8 for more information on these options.) For example, if the LIST Option -gen is specified when calling the Assembler, all NOGEN operands of LIST in the source would be overridden. However, the other operands of LIST in the source would still be effective.

The descriptions of the 10 LIST pseudo operation code options follow:

ON--Turn On Assembly Listing

Resumes printing of the program listing. Since the default is to print the listing, this option normally would be specified only after printing has been turned off by the LIST OFF option.

OFF--Turn Off Assembly Listing

Suppresses the printing of the program listing until the end of code or a LIST ON option is encountered. The default is to print the entire program listing.

COND--Begin Listing False Conditional Assemblies

Lists all blocks of code that are included in the source file as part of a conditional assembly. The blocks are listed regardless of whether or not the IF condition is true or false during assembly.

Since the Assembler normally prints all blocks of conditional assembly, this option has effect only if printing has been turned off with the LIST NOCOND option.

NOCOND--Do Not Print False Conditional Assemblies

The Assembler does not list conditional assembly blocks of code that are not included in the object code because the IF for the block is false. The Assembler still lists blocks that are included in the object file because the IF for the block is true.

This option will remain selected until the end of code or a COND option. The option is turned off when assembly of a program begins and thus must first be selected using the LIST pseudo operation code. Selection of NOCOND in no way affects the object code of an assembled file.

GEN--Begin Listing Generated Macros

Forces the printing of the macro expansion following every macro call, until end of code or a NOGEN option. GEN is the default when assembly of a source file begins; therefore, it would generally be selected only to override a previous NOGEN option.

NOGEN--Do Not Print Generated Macros

Forces the Assembler to not print macro expansions. However, macro definitions are always printed, as are the macro calls themselves; it is only the code that the macro generates that is not printed. This option remains selected until the end of code or a GEN option. The option is turned off when assembly of a program begins and thus must first be selected using the LIST pseudo operation code. Selection of NOGEN in no way affects the object code of generated macros of an assembled source file.

TEXT--Print Object Code

Causes the additional lines created by DC pseudo operation codes to be listed as part of the assembly listing. Since the default is not to print the additional lines, this option may be turned on at the beginning of the assembly.

NOTEXT--Do Not Print Object Code

Suppresses the printing of the additional lines created by DC pseudo operation codes in the assembly listing until the end of code or a LIST TEXT option is encountered.

XREF--Cross-Reference Symbols

The Assembler includes all symbols found after this option in the Symbol Cross-Reference and in the Symbol Table if it is generated as part of the assembly listing. Including symbols in the cross reference is the default when assembly of a source file begins; therefore, it would generally be specified only to override a previous NOXREF option.

NOXREF--Do Not Cross-Reference Symbols

The Assembler does not include any symbols found after this option in the Symbol Cross-Reference and in the Symbol Table if it is generated. This option remains selected until the end of code or an XREF option is found. The default at the beginning of the assembly is to include symbols in the cross-reference.

A typical reason for selecting this option is to prevent the cross-referencing of symbols found in files that are included in the code because of the *INCLUDE and *MACLIB pseudo operation codes when not all of the symbols in the included code are used in the main program.

***MACLIB--INCLUDE MACRO DEFINITION LIBRARY**

Use the *MACLIB pseudo operation code to have the Assembler copy the contents of a macro definition library into the source code. Up to 16 different libraries may be defined during one assembly.

The format is:

```
*MACLIB  'pathname'
```

where:

pathname

is the name of the macro library to be included.

The *MACLIB pseudo operation code must begin with the asterisk in column 1. No label field is permitted with this operation code. The filename may follow the *MACLIB after at least one delimiter (SPACE or TAB).

At present, the *MACLIB pseudo operation code is not implemented.

MACRO--BEGIN MACRO DEFINITION

Use the MACRO pseudo operation code to mark the beginning of a macro definition, to name the macro, and to specify the parameters used within the macro. The names of macros defined with this pseudo operation code are cross-referenced in the symbol cross-reference table.

The format is:

```
name  MACRO[.\parameter0]  [\parameter1,\parameter2,...]
```

where:

name

is a legal Assembler label, as described in Chapter 2. The name is used to call the macro.

\parameter0

is an optional "dummy" name that the Assembler will substitute with a value given as an extension of the macro name in the macro call. The backslash indicates that this is a parameter and must precede the parameter name everywhere it appears in the macro definition.

\parameter1,\parameter2,...

are "dummy" names that the Assembler will substitute with values given in the macro call. The backslash character must precede the parameters both in the MACRO statement and in the macro definition. As many parameters may be named as will fit on one statement. Commas must separate the parameters. A SPACE ends the parameter list.

Parameter names that appear early in the list should not be subsets of parameters that appear later in the list because the Assembler has no way

of determining the end of a parameter. Chapter 3 gives examples of legal and illegal parameters.

Chapter 3 gives a complete description of using and writing macros.

MEND--END MACRO, REPEAT EXPANSION, or STRUCTURE DEFINITION

Use the MEND pseudo operation code to terminate the block of code that forms a macro, repeat expansion, or structure definition.

The format is:

(no label) MEND (no operand)

The rules for using MEND in macro and repeat expansion definitions are given in Chapter 3. The rules for using MEND in structure definitions are discussed in Chapter 5.

NAME--MODULE NAME

Use the NAME pseudo operation code to assign a name to a module for use by the linker. The format is:

(no label) NAME name

where

name

is the name of the module and follows the same syntax rules as those given for labels in Chapter 2.

The NAME pseudo operation code is optional; it is not required for linking of modules. However, if the NAME pseudo operation code is omitted, the Assembler automatically assigns the filename to be the module name. Chapter 4 describes the requirements for naming modules.

NAME is different from TITLE. The TITLE pseudo operation code merely tells the Assembler to print a heading at the top of each page of the listing but has no effect on the object code. NAME forces the name of the module to be saved as part of the object code file.

Thus, a library manager program is able to locate object code files by name.

OPT--CHOOSING THE ADDRESS MODE

To help the Assembler in making the correct choice from various addressing modes, an internal variable whose (bit) structure defines the allowed addressing modes is provided. The structure of that variable, together with a discussion on choosing the optimal mode in a given situation, is described in Chapter 2. The variable containing the mode setting information is initialized to the value OPT_DEFAULT (see the file **/equ/optequ.asm**), with the consequence that:

- All absolute addresses will be long absolute addresses.
- All external references will be translated to absolute addresses.
- All references to other program sections will be translated to absolute addresses.
- Forward references will be translated into program counter (PC) relative addresses if the instruction allows it; otherwise, they are translated into long addresses.
- Absolute values for addresses are considered to be programming errors.

The OPT pseudo instruction has the syntax:

```
OPT    <expression>
```

which sets the above-mentioned internal variable to the value of a given expression. Obviously, the expression should be built using the bit numbers from the **/equ/optequ.asm** file.

POP

To simplify the stack manipulation there are two forms of the POP pseudo operation code, with the syntax:

```
[label:]    POP[.W,.L]    <ea>

[label:]    POP[.W,.L]    <register list>
```

where:

<ea>

is the effective address of the value to be popped from the stack.

<register list>

is a list of two or more registers that are popped from the stack by the MOVEM instruction.

In the first form, the POP instruction is converted into

```
MOVE[.W,.L]     (A7)+,<ea>
```

This means that all forms of the effective address are allowed and the condition code register is affected by the move (unless <ea> is an address register).

In the second form, the register list may be any combination of address and data registers. The condition code register is not affected. The <register list> may be built from individual registers (in any order), separated by the "/" character as in:

```
POP.L     D2/A4/D0
```

or a range of registers as in:

```
POP.W     D1-D4
```

or both as in:

```
POP     D0-D3/A2-A4/D5
```

PSECT--PROGRAM SEGMENT

Use the PSECT pseudo operation code to define the characteristics of a program segment.

The format is:

```
[label:]   PSECT   [name[(<attrib>,<attrib>,...)]]
```

name

identifies the program segment. Name must follow the syntax for label, or it may be written in quotes and then can be an arbitrary string. Each PSECT must have a unique name.

attrib

may be one of the following:

REA	readable
WRI	writable
EXE	executable
SHA	sharable
COM	common
ABS	absolute

The meaning of the attributes is discussed in chapter 4.

The first time a program segment is selected, the PSECT pseudo operation code must define all the attributes. Subsequent usages of the same program segment should just define its name. Note that the assembly starts in the default program segment with no name and attributes REA, WRI and EXE. Additional program segments are assigned attributes REA and WRI unless otherwise specified.

Chapter 4 describes the use of program segments.

PUSH

To simplify the stack manipulation, there are two forms of the PUSH pseudo operation code, with the syntax:

```
[label:]    PUSH[.W,.L]    <ea>

[label:]    PUSH[.W,.L]    <register list>
```

where:

<ea>

is the effective address of the value to be pushed on the stack.

<register list>

is a list of two or more registers that are pushed on the stack by the MOVEM instruction.

In the first form the PUSH instruction is converted into

```
MOVE[.W,.L]    <ea>,-(A7)
```

This means that all forms of the effective address are allowed, and the condition code register is affected by the move (unless <ea> is an address register).

In the second form the register list may be any combination of address and data registers. The condition code register is not affected. The <register list> may be built from individual registers (in any order), separated by the "/" character as in:

```
PUSH.L    D2/A4/D0
```

or a range of registers as in:

```
PUSH.W    D1-D4
```

or both as in:

```
PUSH    D0-D3/A2-A4/D5
```

***RELLIB--INCLUDE LIBRARY OF RELOCATABLE ROUTINES**

Use the relocatable library pseudo operation code to have the linker search a library of relocatable routines for definitions of unresolved global labels.

The format is:

```
*RELLIB 'filename'
```

where:

filename

is the name of the file containing the routines.

***RELOBJ--INCLUDE ANOTHER OBJECT FILE**

Use the force load pseudo operation code to have the linker load some other file.

The format is:

*RELOBJ 'filename'

where:

filename

is the name of the file that the linker will load automatically.

REM--REMARK

Use the REM pseudo operation code to write a comment that will be printed starting in column 1 of the program listing. The REM pseudo operation code itself is never printed.

The format is:

(no label) REM [remark]

where:

remark

is any character string you want printed. It does not have to be enclosed in quotes and can be as long as will fit in one statement.

The Cromemco 3703 Printer will expand a line if the line contains the CONTROL-N (0Eh) character. With CONTROL-N as the first character of a remark, the printer expands the line to make it more noticeable. However, when using this feature, be sure that the remark to be printed does not exceed half the width specification of the WIDTH and TRUNC options of the Assembler call statement. (Refer to Chapter 8.) For example, most listings use the default value of TRUNC=128; thus, the number of characters in the REM pseudo operation code that use CONTROL-N should not exceed 64. This prevents the printer from printing off the paper.

REPT--REPEAT EXPANSION

Use the REPT pseudo operation code to write a repeat expansion definition that repeats the definition a fixed number of times.

The format is:

[label:] REPT expression

where:

label

is an optional label that will be assigned the value of the program counter for the first byte of the first instruction of the first expansion.

expression

is any legal expression, as defined in Chapter 2, that gives a numeric value defining the number of times the definition is to be repeated.

Chapter 3 contains a complete description of repeat expansions.

SET--SET EQUATED VALUE

Use the SET pseudo operation code to assign a value to a label.

The format is:

label SET expression

where:

label

is the label to be equated with the value.

expression

is any legal expression, as defined in Chapter 2, specifying the value. The terms of the expression must have been previously defined. Note that SET may redefine the value of a label (opposite to the EQU pseudo operation code).

Chapter 5 gives examples of the use of equated values.

STRUCT--STRUCTURED EQUATE

Use the STRUCT pseudo operation code to define a series of equated values to be used as offsets from a base.

The format is:

(no label) STRUCT expression

where:

expression

specifies the initial value for the offset. You must give a value, even if it is zero.

As described in Chapter 5, the STRUCT statement is used with DS and MEND statements to define the offsets.

For example, the following structure:

```
          STRUCT  20
X:        DS.B    4
Y:        DS.B    2
Z:        DS.B    1
SIZE:     DS.B    0
          MEND
```

is the functional equivalent of:

```
X:        EQU     20
Y:        EQU     24
Z:        EQU     26
SIZE:     EQU     27
```

Each of these sections of code defines offsets of the labels X, Y, Z, and SIZE.

The DS pseudo operation code does not reserve storage area within the bounds of a structure definition.

SUBTTL--ASSEMBLY LISTING PAGE SUBTITLE

Use the SUBTTL pseudo operation code to print a subtitle at the top of each page of a print listing beginning in column 1. The format is:

```
(no label)  SUBTTL  [subtitle]

            or

(no label)  TITLE2  [subtitle]
```

where:

subtitle

is an optional ASCII character string that does not need to be enclosed in quotes and can be as long as will fit in the statement or on the line of the assembly listing.

As with the REM pseudo operation code, the Subtitle phrase may contain the character CONTROL-N (0Eh). On the Cromemco 3703 Printer, this character expands the line to twice its normal width. For this reason, when using CONTROL-N in a SUBTTL pseudo operation code, the number of characters in the Subtitle phrase should not exceed half the number of characters that will be specified in the WIDTH or TRUNC option of the Assembler call statement (refer to Chapter 8).

The SUBTTL pseudo operation code should be the second line of a program in order to be printed on page 1 as well as on the other pages. Subtitles may be changed in the middle of a source program simply by giving a new SUBTTL pseudo operation code. In this case, SUBTTL causes an automatic formfeed. The Assembler inserts a blank line where the Subtitle phrase would be if SUBTTL is not specified.

TITLE--ASSEMBLY LISTING PAGE TITLE

Use the TITLE pseudo operation code to print a title at the top of each page of a print listing beginning in column 1. The format is:

```
(no label)  TITLE  [title]
```

where:

title

is an optional ASCII character string that does not need to be enclosed in quotes and can be as long as will fit in the statement or on the line of the assembly listing.

As with the REM pseudo operation code, the title may contain the character CONTROL-N (0Eh). On the Cromemco 3703 Printer this character expands the line to twice its normal width. For this reason when using CONTROL-N in a TITLE pseudo operation code, the number of characters in the title should not exceed half the number of characters that will be specified in the WIDTH or TRUNC option of the Assembler call statement (refer to Chapter 8).

The TITLE pseudo operation code should be the second line of a program in order to be printed on page 1 as well as on the other pages. Titles may be changed in the middle of a source program simply by giving a new TITLE pseudo operation code. In this case, TITLE causes an automatic formfeed.

USING

The format is:

[label:] USING <value>, <A-register>

Use the USING pseudo operation code to help the Assembler generate shorter addresses. USING tells the Assembler that the specified address register **<A-register>** is loaded with the declared **<value>** so that it can generate the effective address of an instruction in the form of "address register with displacement."

For the Assembler to generate such an effective address, the address specified in the instruction should be in the same program segment as the declared value and within a range of 32 Kbytes of it. The USING instruction is in effect until the end of the program, or until the corresponding DROP pseudo instruction is encountered. It is the programmer's responsibility to load the A register with the declared value. The USING pseudo instruction merely notifies the Assembler it is there.

VER--VERSION OF THE PROGRAM

The format is:

```
VER  <version>,<revision>

VER  <version>.<revision>
```

may be used to include the version and revision number of the program into the generated object file. **<revision>** and **<version>** may be arbitrary expressions which evaluate to an absolute value in the range 0 to 255.

Chapter 8

CALLING THE ASSEMBLER

Once you have prepared the source file, you can assemble it by calling the Assembler program, ASM. For each source code file listed in your call to the Assembler, ASM:

1. Reads the source code file;

2. Assembles the 68000 source code;

3. Produces a file containing the object code module; and

4. Produces a file containing the listing for the assembly.

The figure also shows the use of file extensions by the Assembler. The source code file must have the extension **.asm**. The object code file is given the extension **.o68**, and the print listing file is given the extension **.prn**.

Chapter 9 describes the error messages that the Assembler can produce.

The format of the Assembler call is:

asm <options and source-code-filenames>

where:

asm

Calls the Assembler.

source-code-filenames

Are the names of one or more files containing source code to be assembled. You can prefix the filenames with pathnames specifying the directories containing the files. Specifying Source Code Files in this chapter gives more detailed information.

options

Are various optional parameters specifying default directories or requesting special options in the physical layout of the print listing, the information to be included in the print listing, or other features. The available options are described in this chapter.

You can mix the options and filenames in any order. Each option applies to the files that follow that option.

SPECIFYING THE SOURCE CODE FILES

When you call the Assembler, you must tell it:

1. The names of the files you want assembled; and

2. The directories in which the files reside.

The filenames must match the names of the files that contain the source code. This means that any name given must contain 1 to 24 characters from the following set:

A-Z a-z 0-9 $ _ .

The source filename will have the extension **.asm.** If you do not write the extension, the Assembler appends it.

You can use ambiguous characters in the filenames in your call to represent strings of characters in the names of files. Say, for example, that a directory contains -- among other files -- three files named **sort1.asm, sort2.asm,** and **sort3.asm.** If you want to assemble all three files, you could list them separately in the call:

asm sort1 sort2 sort3

or use the Cromix Shell ambiguous character expansion:

asm sort?.asm

where **?** matches any single character. For example, suppose a directory contains these files:

```
sortv1.asm  manv1.asm   purv1.asm
sortv2.asm  manv2.asm   purv2.asm
                        purv3.asm
```

and v1, v2, and v3 in the filenames represent different versions of modules. If you want to assemble all the modules that begin with **manv**, you could use this call:

```
asm  manv?.asm
```

The Assembler will assemble every file whose name contains the characters **manv**, followed by a single character, followed by the filename extension **.asm**. In this directory, this would be the files **manv1.asm** and **manv2.asm**.

If you want to assemble only the second version of the modules (containing the string v2) you could use this call:

```
asm  *v2.asm
```

where * matches any string.

The Assembler will assemble every file whose name begins with any number of characters, followed by **v2**, followed by the filename extension **.asm**. In this directory, this would be the files **sortv2.asm**, **manv2.asm**, and **purv2.asm**.

In a similar manner, if you want to assemble all the modules in the directory, you could use this call:

```
asm  *.asm
```

where * matches any string preceding the filename extension **.asm**.

You have three options for specifying the directory in which a file resides:

- You can give a pathname for the directory as part of the filename:

```
asm /assembly/library/sortvl ^/root
```

In this example, the Assembler uses the pathname **/assembly/library** to find the file **sortvl.asm** and uses the pathname **^/** to find the file **root.asm.**

If you give a pathname, the Assembler always looks in the specified directory for the file.

- You can use the -asm parameter to specify a default directory:

```
asm -asm /assembly/manufacturing manv2 purv3
```

In this example, the Assembler looks in the directory **/assembly/manufacturing** for the files **manv2.asm** and **purv3.asm.**

Unless you give a pathname as part of the filename, the Assembler looks in the default directory for a file.

- You can refer to the current directory by specifying neither a pathname nor a default directory:

```
asm *.asm
```

In this example, the Assembler assembles every file having the extension **.asm** that is in the current directory.

You can mix the various forms of specifying a directory:

```
asm /library/sortvl -asm /manufacturing manv*.asm
```

In this example, the Assembler looks for the file **sortvl.asm** in the directory **library** because the pathname is given as part of the filename. The Assembler looks for the files that match **manv*.asm** in the directory **manufacturing** because this is the default directory.

Provided the length of the call does not exceed 128 characters, you can specify any number of files to be assembled with a single command to the Assembler.

SPECIFYING THE DESTINATION OF OBJECT CODE AND PRINT LISTING FILES

The Assembler normally places the object code and print listing files for each module in the same directory from which it read the source code file. You can override this placement with two parameters:

- The -o68 parameter specifies a directory into which all object code files are to be placed, regardless of where the source code files reside. In this example:

  asm -o68 /object /library/sort2 /manufacturing/root

  the object code from the files modules **sort2** and **root** would go in the directory **/object.**

- The -prn parameter specifies a directory into which all print listing files are to be placed, regardless of where the source code files reside. In this example:

  asm -prn /print /library/sort2 /manufacturing/root

  the print listing from the files modules **sort2** and **root** would go in the directory **/print.**

Specifying Options in the Assembler Call

You can specify the following types of options when calling the Assembler:

- Options that specify default directories for the source code files, the object code files, and the print listing files. The options that fall into this group are -asm, -o68, and -prn.

- Options that control the type of information placed in the assembly listing.

  For example, you can include a cross-reference of the program's labels in the assembly listing, while excluding the listing of the program's symbols. The options that fall into this group are -COND, -NOCOND, -GEN, -NOGEN, -LISTON, -LISTOFF, -XREF, -NOXREF, -SYMBOL, -TEXT, and, -NOTEXT.

- Options that control the layout of the assembly listing.

  For example, you can control the width of lines in the listing. The options that fall into this group are -PAGE, -TRUNC, and -WIDTH.

You can specify any number of options in a single Assembler call, as long as the total call is not more than 128 characters. If your terminal automatically generates a RETURN before 128 characters are reached, you can issue a CONTROL-E (a physical but not logical RETURN) to continue the line up to the maximum length. You can specify options in any order, but they must be separated by at least one blank space. You can specify the options in upper- or lower-case letters, or a mixture of the two.

The following sections describe these options.

-ASM - Using the -ASM option specifies a default directory for source code files. The Assembler searches this directory for any file for which you do not specifically give a pathname. Specify the default directory in this way:

```
-ASM pathname
```

The pathname specifies the default directory either by specifying the path from the root directory or from the current directory.

If you do not specify a default directory, the Assembler uses the current directory.

-COND, -NOCOND - If -COND is specified, the Assembler lists all blocks of code that are included in the source file as part of an IF pseudo operation code condition. The blocks are listed regardless of whether or not the IF condition is true or false during the assembly. If -NOCOND is specified, the Assembler does not list any blocks of code that are part of a **false** IF condition.

The -COND and -NOCOND options override any specification made with the LIST pseudo operation code in the source file. If you do not use either option when calling the Assembler, then any specification made with the LIST pseudo operation code takes affect. If

you do not use either the options or LIST, then all blocks of code that are part of an IF condition are included, regardless of whether or not the condition is true or false.

-GEN, -NOGEN - If -GEN is specified, the Assembler lists the source code and object code produced by macro calls in the source file. If -NOGEN is specified, the Assembler does not list the code produced by macro calls. Either way, the macro call itself appears in the assembly listing.

The -GEN and -NOGEN options override any specification made with the LIST pseudo operation code in the source file. If you do not use either option when calling the Assembler, any specification made with the LIST pseudo operation code takes affect. If you do not use either the options or LIST, then the code produced by the macro calls will be shown.

-LISTON, -LISTOFF - If -LISTON is specified, the Assembler lists the source code and object code in the assembly listing. If -LISTOFF is specified, the Assembler does not list these codes.

The -LISTON and -LISTOFF options override any specification made with the LIST pseudo operation code in the source file. If you do not use either option when calling the Assembler, then any specification made with the LIST pseudo operation code takes affect. If you do not use either the options or LIST, the codes are listed.

-PAGE - The -PAGE option controls the number of lines that will appear on each page of the assembly listing. This number includes the space for the heading printed at the top of each page, but does not include the blank lines used to separate one page from another. Use this option if you do not wish to have 59 lines, the default, of listing per page. You may specify as few as 10 or as many as 256 lines per page. The format of this option is:

```
-PAGE lines-per-page
```

-SYMBOL - If -SYMBOL is specified, the Assembler prints a listing of all symbols found in the source file, together with their associated value. If you do not specify -SYMBOL, you do not get this listing.

-TEXT, -NOTEXT - If -TEXT is specified, the Assembler prints all the lines of the object code generated by DB pseudo operation codes in the assembly listing. If -NOTEXT is specified, the Assembler prints only one line of the object code.

The -TEXT and -NOTEXT options override any specification made with the -TEXT, -NOTEXT options of the LIST pseudo operation code in the source file. If you do not use either option when calling the Assembler, any specification made with -TEXT, -NOTEXT in the source file takes affect. If you do not use either the options or pseudo operation codes, only one line of code generated by a DC pseudo operation is listed.

-TRUNC - The -TRUNC option specifies the maximum line length used in the assembly listing. Lines over the specified length are truncated. Use this option if you want to specify a line length other than the default (128 characters) and to truncate lines that exceed the selected length. You can specify a line length between 39 and 255 characters. The format of this option is:

```
-TRUNC  line-width
```

The -TRUNC and -WIDTH options are mutually exclusive. If both are used in the same Assembler call, the last one in the call is used. The -TRUNC option differs from the -WIDTH option in that, if the -TRUNC option is in effect, lines longer than the maximum specified are truncated. If the -WIDTH option is in effect, lines longer than the maximum specified are wrapped around to the next line of the assembly listing.

If you specify neither -TRUNC nor -WIDTH, the Assembler truncates any line over 128 characters.

-WIDTH - The -WIDTH option specifies the maximum line length used in the assembly listing. Lines over the specified length are wrapped around, that is, continued, on the succeeding line. Use this option if you want lines shorter or longer than 128 characters, the default, and you want lines exceeding the length you

select to be wrapped around. You can specify a line length between 39 and 255 characters. The format of this option is:

-WIDTH line-width

The -TRUNC and -WIDTH options are mutually exclusive. If both are used in the same Assembler call, the last one in the call is used. The -TRUNC option differs from the -WIDTH option in that if the -TRUNC option is in effect, lines longer than the maximum specified are truncated. If the -WIDTH option is in effect, lines longer than the maximum specified are wrapped around to the next line of the assembly listing.

If you specify neither -TRUNC or -WIDTH, the Assembler truncates any line over 128 characters.

-XREF, -NOXREF - If -XREF is specified, the Assembler prints a cross-reference of all labels found in the source file as part of the assembly listing. If -NOXREF is specified, the Assembler does not print the cross-reference. The default is to print the cross-reference.

Chapter 9

ERROR MESSAGES

The Assembler generates two kinds of error messages. Some messages inform you of errors in the call to the Assembler, while others note errors in the source code as it is assembled.

ERROR MESSAGES GENERATED FOLLOWING A CALL TO THE ASSEMBLER

When an error is detected in a call to the Assembler, the Assembler aborts and returns control to the Cromix Operating System.

Cannot open input file <pathname>

This message is generated if the specified pathname does not exist or the user does not have the correct access privileges.

Cannot open output file <pathname>

This message is generated if the user does not have the correct access privileges to write into the directory.

Cannot open print file <pathname>

This message is generated if the user does not have the correct access privileges to write the file into the directory.

ERROR MESSAGES GENERATED DURING ASSEMBLY

Error messages generated during assembly of source code inform you of a wide range of incorrect specifications, such as misspelled opcodes, invalid addresses, etc. When an error occurs, the Assembler prints the

applicable error message on the line immediately following the error. The message is a complete expression (not a number), occupying an entire line in the program listing. A copy of the message is also sent to the console.

In most cases, the Assembler, upon encountering an error, assembles the code such that the correct number of bytes are reserved. Thus (if possible) the addresses are numbered correctly, and the assembled instruction is as close to the "correct form" as possible.

A string of asterisks precedes and trails each error message. Following assembly the total number of errors is printed.

Absolute value required

Some instructions require an absolute (i.e. nonrelocatable) expression.

Address mode error

Invalid address mode - may be caused by incorrect OPT setting.

Address register required

Instruction requires an address register.

Address required

Instruction requires an address of some form.

Cannot open include file

Include file was not found, or the user has no privilege to read it.

Data register required

Instruction requires an data register.

EXITM without MACRO

EXITM must be located within a macro body.

Entry/External conflict

Entry point name must not be the same as the name of an external symbol.

Evaluation stack empty

This message signals an error in expression evaluation.

Expression error

Syntax error in expression evaluation.

External symbol not allowed

Instruction does not allow labels declared to be externals.

IFs nested too deep

Too many nested IF statements.

Illegal transfer address

The END pseudo instruction has an illegal transfer address.

Immediate operand needed

Instruction requires an immediate operand.

Label - register name conflict

Register name should not be the same as a label name.

Label not allowed

This pseudo instruction code does not allow the label field.

Label required

This instruction must have the label field.

MACRO not terminated

Macro instruction should be terminated by a MEND pseudo instruction.

MEND without macro

MEND instruction makes sense only within a macro definition.

Macro needs a name

Self-explanatory.

Missing END

Every module must be terminated by an END pseudo instruction.

Multiple definition

A symbol (or a macro name) was defined more than once.

Nested too deep

Macro calls nested too deep.

No comma between arguments

Multiple arguments should generally be separated by a comma.

No matching FI

IFcc without a matching FI.

No matching IF

ENDIF without a matching IF.

No matching IFcc

FI or ELS without a matching IFcc.

Phase error

The Assembler found a different definition for a symbol in the second pass than it found in the first. This indicates an error in the Assembler.

Range error

Expression is too big to be taken as a displacement. The actual limit on the size depends on the instruction.

SET/EQU conflict

A symbol may be used in an EQU statement or in a SET statement but not in both.

Size error

Unacceptable size extension for this instruction.

Syntax error

Syntax error of any kind in an instruction.

Too many '('

Error in expression.

Too many nested IFcc

Self-explanatory.

Too many nested includes

Self-explanatory.

Too many psects

Self-explanatory. (A maximum of 10 psects is allowed.)

Undefined opcode

Unrecognized operation code.

Undefined symbol

A symbol that was not defined was used in the instruction.

Undefined symbol in pass 1

Some pseudo instructions (such as IF, REPT) require expressions that are defined before they are used.

Unknown * directive

The only pseudo instructions starting with an asterisk in column 1 are INCLUDE, RELLIB and RELOBJ.

Unknown PSECT option

Self-explanatory.

Warning: The Assembler has word_aligned the previous instruction

The Assembler skipped a byte to align the location counter to an even address.