

Chapter 4

An Assembler Program

Objectives

This chapter discusses the operation of a typical assembler. We will learn the assembler syntax now to be able to more easily understand examples showing the instruction set in the next chapter.

4.1 Assembly Language Example

In this chapter we learn about some of the features of a typical assembler, at least enough to be able to understand the syntax of the examples in the next chapter. Although this assembler is a component of the Metrowerks CodeWarrior® development system for the Freescale M68HC12 microcontroller, we will not give detailed information on the user interface for operating in that development environment.

We briefly discuss the operation of an *absolute* assembler, the simplest to learn, at least for beginning assembly language programmers. We then transition quickly to the more advanced features of the CodeWarrior development software. We will be able to develop more complex and longer assembly language programs by learning about the *relocatable* assemblers and *linkers*.

An assembler converts source files to machine code, but before we look at how our operates, let us consider a short example.¹ At this stage you probably will not know what the instructions mean or what they do, nor will you understand all that the assembler does. Our goal is to give an overview of the process before we show the component parts of an assembly language program and how the assembler works.

Probably the most famous of all beginning programs, at least for C language programmers, is one that prints “Hello World!”. Example 4-1 is a simple program doing just that.

¹ Chapter 5 in *Microcontrollers and Microcomputers: Principles of Software and Hardware Engineering*, Oxford University Press, NY, 1997, describes how an assembler converts source files to machine code. It also has information to help with debugging and hints for writing assembly language programs.

Example 4-1 Hello World! Example Program

Metrowerks HC12-Assembler
 (c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			; Example program to print "Hello World"
2	2			; Source File: hello.asm
3	3			; Author: F. M. Cady
4	4			; Created: 9/02
5	5			; *****
6	6			; Constant equates
7	7		0000 0000	EOS: EQU 0 ; End of string
8	8			; Linker symbols
9	9			; export 'Entry' symbol
10	10			XDEF Entry
11	11			; Subroutine to printf
12	12			XREF printf
13	13			; Stack pointer
14	14			XREF __SEG_END_SSTACK
15	15			; *****
16	16			Entry:
17	17			; Initialize stack pointer
18	18	000000	CFxx xx	lds #__SEG_END_SSTACK
19	19			; Print Hello World! string
20	20	000003	CCxx xx	ldd #HELLO ; Pass adr of string.
21	21	000006	16xx xx	jsr printf ; Jump to subroutine.
22	22			; Spin forever
23	23			spin:
24	24	000009	20FE	bra spin
25	25			; Define the string to print
26	26	00000B	4865 6C6C	HELLO: DC.B 'Hello World!',EOS
			00000F 6F20 576F	
			000013 726C 6421	
			000017 00	

The sample program in Example 4-1 has several parts. The listing you see is called the *.LST* file and is produced by the HC12 assembler to use when debugging and documenting your work. It shows, from left to right in columns, (1) *Abs.*, the absolute line number for the source code including all include files and macro lines, (2) *Rel.*, the relative line number showing each line in the current source file, (3) *Loc*, the address in memory in which the assembled code is found, (4) *Obj. code*, the assembled code bytes, and finally (5) *Source line*, the source code with label, opcode, operand and comment fields. In this program *lines 1 - 4* are comments that introduce the program. *Line 7* uses an *assembler directive*, called an *equate* or *EQU*, to define the value used for the *symbol EOS*. This defines the code-byte that signifies the end of the message to be printed on the terminal (such as the null byte at the end of a string in a C program). *Line 8 - 14* define symbols to be used by the linker including the *Entry* point and a subroutine, *printf* which is to do the actual printing. The actual program code appears in *lines 17 to 24*. The stack pointer register is initialized (*line 18*), the message is printed (*lines 20 - 21*) and the program terminates by entering a spin loop where it stays until the microcontroller is reset. At the bottom of the program (*line 36*), an assembler directive, *DC.B*, defines the message *Hello World!*

This is a complete assembly language program for the M68HC12, and you will be producing programs that look very much like this. In the following sections we will examine each component part of a program and describe the operation of the HC12 assembler.

4.2 Metrowerks HC12 Assembler

The Freescale HC12 assembler converts M68HC12 assembly language source files into *S Record* files that can be loaded into the microcontroller's memory.

The Metrowerks HC12 assembler runs on an IBM compatible personal computer (and other platforms) and *cross-assembles* code for the M68HC12 microcontroller. It can produce a variety of output files and we will start using what is known as an *S-Record* file that is loaded into the M68HC12 memory.

specify operations and operands. You will also learn about assembler pseudo-operations and directives that help the assembler do its job and make programs easier for us to read.

HC12 can operate as an *absolute* or a *relocatable* assembler.

The assembler converts the program's operation mnemonics to opcodes and its operands to operand codes. Your primary task at this time is to learn the syntax of this assembler to be able to

When you are using an *absolute assembler*, all source code for the program must be in one file or group of files assembled together. This is the easiest way to operate and we will use it to begin your programming chores. As your programs grow, however, we will change to relocatable assembler mode.

4.3 Assembler Source Code Fields

Each source code line has four fields - label, operation mnemonic, one or more operand, and comment, as shown in Table 4-1. The fields are separated by a white space (usually one or more tab characters, shown as <tab> so the fields line up) and there are specific rules for each field.

Table 4-1 Source Code Fields

Label Field	Opcode Field	Operand(s) Field	Comment Field
Example: <tab>	Ldaa	<tab> #64	<tab> ; Initialize word counter

Label Field

The *label field* starts in the first column of the source code line.

The *label field* is the first field of a source statement. A label is a symbol followed (optionally) by a colon. The label is optional but when used can provide a symbolic memory reference, such as a branch instruction address, or a symbol for a constant. A valid label is:

- Alphanumeric characters, which may be upper or lower case letters a-z, digits 0-9, underscore (_) or period (.)
- A label must start with an alphabetic character.
- The label must start in the first column of the source code line unless it ends with

a colon.

- Upper and lower case characters are distinct by default but case sensitivity may be turned off with the assembler `-Ci` switch case sensitivity option. When this is active; the assembler treats lower case the same as upper case.
- A label may end with a colon (:).
- A label may appear on a line by itself.
- Long labels are truncated to ??? characters.

A *whitespace* is a space or <Tab> character.

A whitespace character (blank or <tab>) must be in the first character position in the line when there is no label, and there must be a white space between the label and the following opcode. See Example 4-2 for different kinds of labels.

A label may not occur more than once in your program. If it does, the assembler will give an error message noting that the symbol has been redefined. Remember also that a label must start with an alphabetic character and must start in the first column.

Example 4-2 Labels

```

; Labels Examples
TEST:      ; Legal label
Test:      ; Different label than TEST
_TESTT:    ; Legal label too
Test:      ; Illegal - Duplicate label
TestData:  ; Legal
Test_Data: ; Legal, more readable
Label      ; Legal, a label doesn't need a colon
Label2     ; unless it doesn't start in the first column.

```

Opcode or Operation Field

The *opcode field* begins after the first whitespace character.

The *opcode* field contains either a *mnemonic* for the operation, an *assembler directive* or *pseudo-operation*, or a *macro name*. It must be preceded by at least one white space. The assembler is insensitive to the case of the mnemonic; all upper case letters are converted to lower case. See Example 4-3.

Example 4-3 Operation Field

Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj. code	Source line

1	1			; opfield.asm
2	2	000000	87	CLRA ; Legal mnemonic
3	3	000001	87	clra ; Mnemonics are not case sensitive
4	4			clra ; Not legal, there has to be
5	5			; at least one white space in
6	6			; front of the mnemonic
7	7	000002	08	DC.B \$08 ; Legal assembler directive

Operand Field

The *operand field* follows the opcode with at least one whitespace character between.

The assembler uses the *operand* field to produce the binary code for the operand, and the interpretation of this field depends to a certain extent on the opcode. The operand must follow the opcode and be preceded by at least one whitespace. Operands can be the *symbols*, *constants*, or *expressions* that are evaluated by the assembler. The operand field also specifies the addressing mode for the instruction as shown in Table 4-2.

Table 4-2 Operand formats and addressing modes

Operand Format	Addressing Mode/Instruction Type
No operand	Inherent
Expression	Direct, Extended, or Relative
#Expression	Immediate ²
Expression,R	Indexed offset with X, Y, SP, or PC
Expression,-R	Indexed auto pre-decrement
Expression,+R	Indexed auto pre-increment
Expression,R-	Indexed auto post-decrement
Expression,R+	Indexed auto post-increment
Accumulator,R	Indexed accumulator
[Expression,R]	Indexed indirect
[D,R]	Indexed indirect D accumulator
Expression,Expression	Bit set or clear
Expression,R,Expression	Bit set or clear
Expression,Expression,Expression	Bit test and branch
Expression,R,Expression, Expression	Bit test and branch

Symbols: A symbol represents an 8-, or 16-bit integer value that *replaces* the symbol during the assembler's evaluation of the operand. For example, if the symbol CRLF is defined as \$0D0A, the assembler replaces each occurrence of CRLF in your program with \$0D0A. Special symbols are the asterisk (*) and dollar sign (\$) that represent the current 16-bit value of the location (program) counter.

A symbol may be defined in another program module. In that case it is called an *external* symbol and must be “defined” in the current program as being external by using the *XREF* directive. We will show how this is done later in this chapter when we cover the operation of the relocatable assembler.

The default base for numbers is *decimal*.

Constants: Constants are numerical values that do not change during the program. Constants may be specified in one of four formats - decimal, hexadecimal, binary, or ASCII characters or strings. The format indicators shown in Table 4-3 can be given as a prefix to indicate the base of the number.³

² It is excruciatingly important that you remember to include the # when you want immediate addressing mode.

³ The assembler can be configured to be compatible with the MCUAsm and Avocet number base designators. Please refer to the full Freescale HC12 assembler manual.

Table 4-3 Base Designators for Constants

Base	Prefix
Binary (2)	%
Decimal (10)	None (default)
Hexadecimal (16)	\$
Octal	@

The default base is normally decimal and is chosen if no other format specifier is given. The default base for the assembler can be changed by the *BASE* directive.

Decimal Constants: The decimal constant is specified no suffix. In lines 4 and 5 in Example 4-4 the data value to be loaded into the register is a decimal value. See Example 4-15 to see how to use the *define constant byte (DC.B)* directive to define decimal constants that can be stored in memory.

Hexadecimal Constants: Hexadecimal numbers are identified with the prefix \$. Hexadecimal values are a string of digits from the hexadecimal symbol set (0-9, A-F). See Example 4-4. Hexadecimal constants are used more frequently in assembly language programs than decimal constants, particularly when specifying addresses. However, if it makes sense to write a decimal constant, do not convert the decimal value to hexadecimal. Write it as a decimal constant and let the assembler convert it. Inspect the *Obj. code* in Example 4-4 to see that a constant, such as 100_{10} can be specified as either a decimal, hexadecimal or a binary value. You should chose the one that makes the most sense to you when you read the program.

Binary constants: Binary constants are specified by the percent sign (%) prefix and are comprised of 1's and 0's. See Example 4-4. Use binary constants to make programs more readable. Suppose you wanted to define a mask for the four least significant bits of a byte; using %00001111 is more readable than using hexadecimal \$0F and is far better than decimal 15.

Example 4-4 Decimal, Hexadecimal and Binary Constants

Metrowerks HC12-Assembler
 (c) COPYRIGHT METROWERKS 1987-2003

Abs. Rel.	Loc	Obj. code	Source line
-----	-----	-----	-----
	1		; constants.asm
	2		; Decimal Constants
	3		; Op Operand Result of instruction
	4	4 000000 8664	ldaa #100 ; A = 100
	5	5 000002 CE04 D2	ldx #1234 ; X = 1234
	6		; Hexadecimal Constants
	7	7 000005 8664	ldaa #\$64 ; A = \$64 = 100
	8	8 000007 869C	ldaa #\$9c ; A = \$9c = -100
	9	9 000009 CE12 34	ldx #\$1234 ; X = \$1234 = 4660
	10		; Binary Constants
	11	11 00000C 8664	ldaa #%01100100 ; A = \$64 = 100
	12	12 00000E 86F0	ldaa #%11110000 ; Most significant
	13		; nibble mask
	14	14 000010 CE12 34	ldx #%0001001000110100 ; X = \$1234

ASCII constants: Single *ASCII constants* or *strings* of one or more ASCII characters are enclosed in single (' ') or double quotation marks (" "). Single quotes are allowed only in strings delimited by double quotes. Double quotes can only appear in single-quote delimited strings. The assembler can assign the ASCII code for any printable character. Use this feature to specify ASCII characters instead of writing the hexadecimal code. The assembler will always make the conversion from the character to the code correctly. It is better to specify 'A' than \$41, although they are equivalent. See Example 4-5 and Example 4-16.

Example 4-5

Show four ways to specify the code for the ASCII code for the character C and choose the best way to load that code into the accumulator A in a program.

Solutions:

ASCII -- 'C' or "C"
 Hexadecimal -- \$43
 Decimal -- 67
 Binary -- %01000011

The best way to load accumulator A with the ASCII code for the character C is

```
ldaa #'C'
```

Expressions make programs more readable and easier to use in other applications.

Expressions: An expression is a combination of symbols, constants, and algebraic operators. The assembler evaluates the expression to produce a value for the operand. HC12 Assembler algebraic operators are shown in Table 4-4.

Expressions are evaluated with a normal algebraic operator precedence that can be altered by using parenthesis. Because expressions are evaluated by the assembler, they may be used for constants only. Nevertheless, the use of expressions is very powerful and can make a program more readable. It can also make it more portable and useful in other applications. See Example 4-6 and Example 4-7.

The assembler allows *binary relational operators*. These compare two operands and return '1' if the condition is true or '0' if the condition is false. These are used most often for conditional assembly. See Example 4-8 and section 0.

Table 4-4 Assembler Expressions

+	Addition	-	Subtraction
*	Multiplication		
/	Division produces truncated result	%	Modulo division
>>	Shift right	<<	Shift left
&	Bitwise AND		Bitwise OR
^	Bitwise Exclusive OR (XOR)	~	1's complement
!	Logical NOT		
!= or <>	Not equal	= or ==	Equal
<=	Less than or equal	<	Less than
>=	Greater than or equal	>	Greater than
HIGH	High byte of an address	LOW	Low byte of an address
PAGE	Page byte of an address		

Example 4-6 Expressions

```
expressions.asm
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003
```

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			TITLE 'expressions.asm'
2	2			; Test of all expression operators
3	3		0000 0001	One: EQU 1
4	4		0000 0002	Two: EQU 2
5	5		0000 00FF	Small: EQU \$ff
6	6		0000 1234	Adr: EQU \$1234
7	7		0002 3456	Adr1: EQU \$23456
8	8			;
9	9	000000	03	Add: DC.B One + Two ; Addition
10	10	000001	01	Sub: DC.B Two - One ; Subtraction
11	11	000002	FF	SUB: DC.B One - Two ; Subtraction
12	12	000003	04	Mul: DC.B Two * Two ; Multiplication
13	13	000004	02	Div: DC.B Two / One ; Division
14	14	000005	00	DIV: DC.B One / Two ; Division
15	15	000006	01	DiV: DC.B One % Two ; Modulo division
16	16	000007	7F	Shr: DC.B Small >> 1 ; Shift right one bit
17	17	000008	0F	SHR: DC.B Small >> 4 ; Shift right 4 bits
18	18	000009	08	Shl: DC.B Two << 2 ; Shift left two bits
19	19	00000A	02	And: DC.B Small & Two ; Bitwise AND
20	20	00000B	03	Or: DC.B Two One ; Bitwise OR
21	21	00000C	FD	Xor: DC.B Small ^ Two ; Bitwise XOR
22	22	00000D	FD	Cmpl: DC.B ~ Two ; 1's Complement
23	23	00000E	12	Upper: DC.B HIGH (Adr) ; High byte of adr
24	24	00000F	34	Lower: DC.B LOW (Adr) ; Low byte of adr
25	25	000010	02	Pg: DC.B PAGE (Adr1) ; Page byte of adr1

Example 4-7 Using an Assembler Expression

Assume an assembler program with two data buffers with the start of each signified by the labels `Data_1` and `Data_2`. The two buffers are sequential in memory and the amount of data in each buffer changes in programs for different applications. Assume that somewhere in your program you want to load accumulator B with the number of bytes in the `Data_1` buffer. Use an expression to do that.

Solution:

Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			; expr1.asm
2	2	000000	C664	ldab #(Data_2 - Data_1)
3	3			; Immediate addressing
4	4			; The assembler computes the difference between
5	5			; the address of Data_2 and Data_1
6	6			;
7	7	000002		Data_1: DS.B 100 ; Allocate 100 bytes
8	8	000066		Data_2: DS.B 2*100 ; Allocate 200 bytes

Example 4-8 Relational Operators

relational.asm
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			TITLE 'relational.asm'
2	2			; Relational expressions
3	3		0000 0001	One: EQU 1
4	4		0000 0002	Two: EQU 2
5	5			;
6	6	000000	00	Equal: DC.B One = Two ; Equal
7	7	000001	01	EQUAL: DC.B One = One ; Equal
8	8	000002	01	NotEq: DC.B One != Two ; Not equal
9	9	000003	01	Less: DC.B One < Two ; Less than
10	10	000004	00	Greater:DC.B One > Two ; Greater than
11	11	000005	01	LTEQ: DC.B One <= Two ; Less or equal
12	12	000006	00	GTEQ: DC.B One >= Two ; Greater or equal
13	13	000007	01	Not: DC.B !0 ; Logical not
14	14	000008	00	NOT: DC.B !Two

Comment Field

The last field in the source statement is the *comment*. Comments start with a semicolon (;) and comments can be a complete line. Any line starting with a (;) or a (*) in column 1 is a comment line. The source program may have blank lines also.

4.4 Assembler Control

Assembler *directives* instruct the assembler how to do its job.

The assembler is controlled by directives (sometimes called *pseudooperations*) that you place into your program. These are an important and vital part of an assembler program. Assembler directives are like opcode mnemonics because they appear in the opcode field, but they are not part of the microcontroller's instruction set. Directives can *define* the program's *location* in memory so all memory addresses are correct. They allow *symbols* and the *contents of memory* locations to be defined. Assembler directives also allocate memory locations for variable data storage. In short, directives help the assembler generate code for the program. Assembler directives also control how the assembler creates its output files, especially the list file. Table 4-5 shows directives available in the HC12 assembler.⁴

Table 4-5 Metrowerks HC12 Assembler Directives

Section	Definition
	ORG Set program counter to the origin of the program in an absolute assembler mode.
	SECTION Define a relocatable section.
	OFFSET Define an offset section.
	Constant Definition
	EQU Equate symbol to an expression (cannot be redefined).
	SET Assign a name to an expression (can be redefined).
	Reserving or Allocating Memory Locations
	DS Define Storage.
	Defining Constants in Memory
	DC.B Define byte constant.
	DC.W Define word constant.
	DCB Define a constant block.
	RAD50 RAD50 encoded string constants.
	Export or Import Global Symbols
	ABSENTRY Specify the entry point in an absolute assembly file.
	XDEF Make a symbol public (visible to some other file).
	XREF Import reference to an external symbol.
	XREFB Import reference to an external symbol located on the direct page.
	Assembly Control
	ALIGN Define alignment constraint.
	BASE Specify default base for constants.
	END End of assembly unit.
	EVEN Define two byte alignment constraint.
	FAIL Generate user defined error or warning messages.
	INCLUDE Include text from another file.
	LONGEVEN Define four byte alignment constraint
	Repetitive Assembly Control
	FOR Repeat assembly blocks.
	ENDFOR End of FOR block.

⁴ In the syntax discussion for each of the assembler directives, the following notation is used:

[] Parenthesis denote an optional element.

<> Angle brackets enclose a syntactic variable to be replaced by a user-entered value.

Table 4-5 Metrowerks HC12 Assembler Directives

Listing Control	
CLIST	Include conditional assembly block.
LIST	Specify that all following assembly lines are in the list file.
LLEN	Define line length.
MLIST	Include macro expansions.
NOLIST	Specify that all following assembly lines are not in the list file.
NOPAGE	Disable pagination in the list file.
PAGE	Insert page break.
PLEN	Define page length.
SPC	Insert empty or blank line.
TABS	Define number of characters to insert for the <tab>.
TITLE	User defined title.
Macro Definition	
ENDM	End of user defined macro.
MACRO	Start of user defined macro.
MEXIT	Exit from macro expansion.
Conditional Assembly	
ELSE	Alternate block, code included if IF statement not true.
ENDIF	End of conditional block.
IF	Start of conditional block.
IFC	Test if two string expressions are equal.
IFDEF	Test if a symbol is defined.
IFEQ	Test if an expression is null.
IFGE	Test if an expression is greater than or equal to 0.
IFGT	Test if an expression is greater than 0.
IFLE	Test if an expression is less than or equal to 0.
IFLT	Test if an expression is less than 0.
IFNC	Test if two string expressions are different.
IDNDEF	Test if a symbol is undefined.
IFNE	Test if an expression is not null.

4.5 Assembler Directives

Section Definition

ORG is used to locate sections of the program in the correct type of memory.

ORG (Set Program Counter to Origin): The **ORG** directive changes the assembler's location counter to the value in the expression. An **ORG** defines where your program is to be located in the various sections of ROM and RAM memory and is

used in *absolute* assembly programs. In *relocatable* assembly programs the *linker* program provides this location function. See Example 4-9 for an absolute code example. See section **Error! Reference source not found.** for a discussion on the relocatable assembler.

```
ORG      <Expression>    [ ; Comment ]
```

Example 4-9 ORG - Set Program Counter to Origin for Absolute Assembly

Metrowerks HC12-Assembler

(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj.	code	Source line
1	1				; org.asm
2	2		0000	F000	ROM: EQU \$F000 ; Location of ROM
3	3		0000	0800	RAM: EQU \$0800 ; Location of RAM
4	4		0000	0A00	STACK: EQU \$0a00 ; Location of stack
5	5				;
6	6				ORG ROM ; Set program counter
7	7				; to ROM for code
8	8				; The following code is located at memory
9	9				; address ROM
10	10	a00F000	CF0A	00	lds #STACK ; Initialize SP
11	11	a00F003	B608	00	ldaa Data_1 ; Load from memory
12	12				; address RAM
13	13				; - - -
14	14				ORG RAM ; Set program counter
15	15				; to RAM for the data
16	16	a000800			Data_1: DS.B \$20 ; Set aside \$20 bytes

SECTION (Declare Relocatable Section): This directive establishes a relocatable section and initializes the assembler's location counter to zero for the first SECTION directive. Any subsequent SECTION directives for that section restores the location counter to the value that follows the address of the last code in the section.

SECTION is used to define sections of the program that will be located *later* by the linker program.

```
<name>: SECTION [SHORT] [<number>]
```

<name> is the name assigned to the section. You may have multiple SECTION directives with the same name to refer to the same section.

<number> is optional and is only specified for compatibility with the MCUASM assembler.

SHORT is an optional qualifier so that you can specify a short section small enough to be located in the base page for using direct addressing.

Sections may be *code*, *constant*, or *data* sections. A code section is one that contains at least one assembly instruction. Constant sections contain only Define Constant (DC) or Define Constant Block (DCB) directives and data sections contain at least a Define Storage (DS) directive. In your embedded application, code and constant sections must be *located* in read-only memory (ROM) and data sections in read/write (RAM) memory. See Example 4-10.

Example 4-10 SECTION

Metrowerks HC12-Assembler

(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			; section.asm
2	2			RAM: SECTION ; Define the section that
3	3			; will go into RAM for variable
4	4			; storage
5	5	000000		Buf_1:DS 10
6	6			
7	7			ROM: SECTION ; Define the section that goes
8	8			; in ROM for the code
9	9			entry:
10	10	000000	B6xx xx	ldaa Buf_1
11	11	000003	7Axx xx	staa Buf_1+1

OFFSET (Create Absolute Symbols): OFFSET declares a section and initializes

An *OFFSET* section may be used to simulate a data structure.

the location counter to the value in <expression>. An OFFSET section allows you to create and access data elements in, for example, a structure. See Example 4-11.

```
OFFSET    <expression>
```

Example 4-11 Using OFFSET to Define a Structure

Metrowerks HC12-Assembler

(c) COPYRIGHT METROWERKS 1987-2003

```

Abs.Loc   Obj. code Source line
-----
1          ; structure.asm
2          ; Example code showing OFFSET directive
3          ; to create and access data structures.
4          ; 6/04
5          ;
6          0000 4000 ROM:    EQU    $4000    ; ROM location
7          0000 6000 RAM:    EQU    $6000    ; RAM location
8          0000 8000 STACK:  EQU    $8000    ; Stack pointer location
9          ; Define the structures
10         ;           OFFSET 0
11         ; Structure 1
12000000   Count1: DS.B  1          ; An 8-bit counter
13000001   Value1: DS.W  1          ; The 16-bit value
14         0000 0003 Size1:  EQU    *          ; This defines the size
15         ;                                     ; of the 1st structure
16         ;           OFFSET 0
17         ; Structure 2
18000000   Count2: DS.W  1          ; A 16-bit counter
19000002   Value2: DS.W  1          ; 16-bit value
20         0000 0004 Size2:  EQU    *          ; Size of second structure
21         ;           ORG    ROM
22004000   CF80 00          lds    #STACK ; Initialize stack pointer
23         ;           - - -
24004003   CE60 03          ldx    #Struct2; Point to 2nd structure
25004006   CD00 00          ldy    #0
26004009   6D00            sty    Count2,x; Init counter
2700400B   ED02            ldy    Value2,x; Get the current value
2800400D   02              iny           ; Increment it
2900400E   6D02            sty    Value2,x; Save it again
30004010   6200            inc    Count2,x; Increment the counter
31         ;           - - -
32         ;           ORG    RAM
33006000   Struct1:DS.B  Size1      ; Define the first structure
34006003   Struct2:DS.B  Size2      ; Define the second
35006007   Data_1: DS.B  6          ; A data buffer

```

Constant Definition**EQU (Equate a Symbol to a Value):** EQU is probably used more than any other

An EQU may be the most used assembler directive.

directive in assembly language programming because it is good programming practice to use symbols where constants are required. Then, if the constant needs to be changed, only the

equate is changed. When the program is reassembled, all occurrences of constants are changed.

```
<Label:>    EQU    <Expression>    [; Comment]
```

Any constant value can be defined for the assembler using the EQU. The EQU directive must have a label and an expression. It is a useful documentation technique to have a comment with each EQU to tell the another programmer reading your program what the symbol is to be used for. An EQU does not generate any code that is placed into memory. See Example 4-12. The default base used to evaluate expressions and other data

is decimal, but you can change the default with the BASE directive.

Example 4-12 EQU - Equate Symbol

Metrowerks HC12-Assembler

(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj. code	Source line
1	1			; equ.asm
2	2		0000 0D0A	CRLF: EQU \$0D0A
3	3			; For each occurrence of
4	4			; CRLF, the assembler will
5	5			; substitute the value \$0D0A
6	6		0000 0005	COUNT: EQU 5
7	7			; Loop counters often need to be initialized
8	8		0000 0019	COUNT1: EQU COUNT*5
9	9			; The assembler can evaluate an expression to
10	10			; provide a value of 25 for COUNT1
11	11		0000 000F	LS_MASK: EQU \$0F
12	12			; A mask that picks off the least significant
13	13			; nibble in a byte
14	14		0000 000F	ls_mask: EQU %00001111
15	15			; A binary mask equate is more readable and
16	16			; informative than one given in hexadecimal
17	17			; Here some code examples using the EQUs
18	18	000000	0D0A	DC.W CRLF
19	19	000002		DS.B COUNT1
20	20	00001B	8605	ldaa #COUNT
21	21	00001D	840F	anda #ls_mask
22	22	00001F	C40F	andb #LS_MASK

SET (Set a Symbol to a Value): SET is similar to the EQU directive except the value set is temporary in that it can be redefined later in the program. Any symbol defined by an EQU cannot be redefined.

```
<Label:> SET <Expression> [; Comment]
```

Reserving or Allocating Memory Locations

The *Define Storage* directive is used to allocate memory for variable data storage.

DS (Define Storage): The DS sets aside memory locations by incrementing the assembler's location counter by the number of bytes specified in the expression. The block of memory reserved is NOT initialized with any value.

```
[Label:] DS[.<size>] <n> [; Comment]
```

Use this directive to allocate storage for variable data areas in RAM and then initialize the variables, if required, in the program at run-time. See Example 4-13 and Example 4-14.

Example 4-13 DS - Define Storage

Metrowerks HC12-Assembler
 (c) COPYRIGHT METROWERKS 1987-2003

Abs.Loc	Obj. code	Source line
1		; ds.asm
2	0000 0010	COUNT_3: EQU \$10
3000000		BUFFER: DS COUNT_3 ; Allocates \$10 bytes
4000010		BUFFER1: DS.B 2*COUNT_3 ; Allocates \$20 bytes
5000030		BUFFER2: DS.W COUNT_3 ; Allocates \$10 words

Example 4-14

Show how to use the DS directive to reserve 10 bytes for data. Initialize each byte to zero in a small program segment.

Solution:

Metrowerks HC12-Assembler
 (c) COPYRIGHT METROWERKS 1987-2003

Abs.Loc	Obj. code	Source line
1		; dsex1.asm
2	0000 000A	NUMBER: EQU 10 ; Number of bytes allocated
3	0000 0800	PROG: EQU \$0800 ; Program location
4	0000 0900	RAM: EQU \$0900 ; Location of RAM
5		ORG PROG
6		; - - -
7000800	C60A	ldab #NUMBER ; Initialize B with a loop
8		; counter
9000802	CE09 00	ldx #BUF ; X points to the start of the
10		; buffer
11000805	6930	loop: clr 1,x+ ; Clear each location and
12		; point to the next location
13000807	0431 FB	dbne b,loop ; Decrement the loop counter
14		; and branch if the loop
15		; counter is not zero
16		; - - -
17		ORG RAM ; Locate the data area
18000900		BUF: DS NUMBER ; Allocate the data area

Defining Constants in Memory

The following pseudo-operations define constants for ROM. We recommend highly that you do not use them to initialize variable data areas in RAM. RAM data areas should be *allocated* with the DS and then *initialized* at run time as shown in Example 4-14.

Strings of ASCII characters may be defined with the *DC.B* directive.

DC (Define Constant): The Define Constant directive allocates memory locations and assigns (initializes) values to each.

[<label>:] DC[.<size>] <expression>[,<expression>, ...]

You can define constants of different sizes, including bytes, words (two bytes), and long words (four bytes). These size definitions are given by the following:

- **DC.B:** One byte is allocated for each expression. If the expression is an ASCII

string, one byte is allocated per character.

- **DC.W:** Two bytes are allocated for numeric expressions. ASCII strings are right aligned on a two-byte boundary.
- **DC.L:** Four bytes are allocated for numeric expressions and ASCII strings are right aligned on a four-byte boundary.

If <size> is not given, the default is for a byte to be allocated and defined. See Example 4-15 and Example 4-16.

A block of memory can be initialized using the *DCB*

DCB (Define Constant Block): You can allocate and initialize a block of memory with the DCB directive.

```
[<label>:]    DCB[.<size>] <count>,<value>
```

Define Constant <size> may be B, W, or L with B the default. The <count> defines the number of elements to be defined and may range from 1 to 4096. The value stored in each location is the sign-extended expression <value>. See Example 4-15.

Example 4-15 Define Constant

Metrowerks HC12-Assembler

(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj.	code	Source line
1	1				; dc.asm
2	2				; Define decimal constants
3	3	000000	64		DC.B 100 ; Define a byte
4	4	000001	0064		DC.W 100 ; Define a word
5	5				; Hexadecimal constants
6	6	000003	23		DC.B \$23
7	7	000004	1234		DC.W \$1234
8	8	000006	ABCD		DC.W \$abcd
9	9	000008	ABCD		DC.W \$ABCD ; Upper case same as lower
10	10				; Binary constants
11	11	00000A	05		DC %0101 ; Valid (.B is default size)
12	12	00000B	65		DC.B 0101 ; Invalid, missing %.
13	13				; Assembler thinks this is
14	14				; decimal 101.
15	15				; Initialize four memory locations with the data
16	16				; 01, 02, \$10, and \$ff
17	17		0000	00FF	MAX: EQU 255
18	18		FFFF	FF80	MID: EQU -128
19	19	00000C	0102	10FF	Data: DC.B 01, 02, \$10, MAX, MID
		000010	80		
20	20				; Initialize four locations with \$ff using
21	21				; the define constant block directive
22	22	000011	FFFF	FFFF	DCB.B 4,MAX
23	23				; Initialize four words with -128
24	24	000015	FF80	FF80	DCB.W 4,MID
		000019	FF80	FF80	

Example 4-16 ASCII Strings

Metrowerks HC12-Assembler

(c) COPYRIGHT METROWERKS 1987-2003

```

Abs.Loc   Obj. code Source line
-----
1         ; ascii.asm
2000000 4161 62          DC.B  "A",'a',"b"
3000003 5468 6973          DC.B  "This is a string."
000007 2069 7320
00000B 6120 7374
00000F 7269 6E67
000013 2E
4         0000 000D CR:    EQU   $0d ; ASCII code for carriage return
5         0000 000A LF:    EQU   $0a ; ASCII code for line feed
6000014 4865 7265 Msg:    DC.B  "Here is a string with",CR,LF
000018 2069 7320
00001C 6120 7374
000020 7269 6E67
000024 2077 6974
000028 680D 0A
700002B 6361 7272          DC.B  'carriage return and line feed '
00002F 6961 6765
000033 2072 6574
000037 7572 6E20
00003B 616E 6420
00003F 6C69 6E65
000043 2066 6565
000047 6420
8000049 6368 6172          DC.B  "characters."
00004D 6163 7465
000051 7273 2E
9000054 00              DC.B  0   ; Null terminator for printf

```

RAD50 (Rad50 Encoded String Constants): This directive places strings encoded with the RAD50 encoding scheme into constants. This encoding places three string characters of a reduced character set into two bytes, thereby saving memory. Only 40 different character values are supported and strings have to be decoded before they can be used.⁵

Export or Import Global Symbols

ABSENTRY (Application Entry Point): ABSENTRY is used when the absolute assembly option is being used. It creates an entry in the absolute code file (.abs) that is used by the debugger.

```
ABSENTRY <label>
```

The following directives, XDEF, XREF, and XREFB, are used when the relocatable assembler features of the Metrowerks HC12 assembler is being used.

⁵ Need to find a reference to RAD50.

XDEF (External Symbol Definition): XDEF allows you to define a label in the current module to be made visible (public, or global) in other modules.

In a *relocatable* assembler, a symbol may be defined in another module than the one currently being assembled by using *XDEF* and *XREF*.

```
XDEF [.<size>] <label>[,<label>] . . .
```

The default <size> is W.

XREF (External Symbol Reference): XREF is the “other hand” of XDEF. It declares, for the current module, that a variable used in this module is defined in another module. If you use an XREF in a module, there must be an accompanying XDEF in some other module.

```
XREF [.<size>] <label>[,<label>] . . .
```

Again, the default <size> is W.

XREFB (External Reference for Symbols Located on the Direct Page): XREFB is similar to XREF except that symbols enumerated here may be located on the base page in another module. This allows direct addressing to be used.

```
XREFB <symbol>[,<symbol>] . . .
```

Assembly Control

The assembly control directives give you control over how the assembler operates.

ALIGN (Align Location Counter): The ALIGN directive forces the next instruction to a boundary that is a multiple of <n>.

```
ALIGN <n>
```

The value of <n> must be between 1 and 32767. Any bytes that are needed to fill the alignment block are initialized with 0.

BASE (Set Number Base): The default base when specifying constants is decimal.

The default base for numbers is decimal but this may be changed by the *BASE* directive.

This can be changed with the BASE directive.

```
BASE <n>
```

The base is set by <n> which may be 2, 8, 10 or 16.

END (End Assembly): This directive causes the assembler to terminate assembling code and any subsequent statements are ignored.

```
END
```

EVEN (Force Word Alignment): The even directive forces the next instruction to the next even address relative to the start of the section.

```
EVEN
```

FAIL (Generate Error Message):

```
FAIL <arg> | <string>
```

??? Do I need to include this???

INCLUDE (Include Text from Another File): Another file can be inserted in the source input stream.

Parts of your source file may be kept in another file called an *include* file.

```
INCLUDE <file specification>
```

The assembler attempts to open the file in the current working directory. If it is not found there, it searches for the file in every path specified in the environment variable GENPATH. The file specification must be enclosed in quotation marks. Include files are often used to define register addresses and other constants.

Example 4-17 INCLUDE

The source file is:

```
; include.asm
;
;           INCLUDE "boilerplate.inc"
; Other code follows
;   - - -
```

and the assembler list file shows the file with the include file.

```
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003
```

Abs.	Rel.	Loc	Obj.	code	Source	line
----	----	----	-----	-----	-----	-----
1	1				; include.asm	
2	2				;	
3	3				INCLUDE "boilerplate.inc"	
4	1i				; An include file	
5	2i				; boilerplate.inc	
6	3i				; This file contains a variety of commonly	
7	4i				; used equates.	
8	5i		0000	000D	CR: EQU \$0d ; ASCII for carriage return	
9	6i		0000	000A	LF: EQU \$0a ; Line feed	
10	7i		0000	0000	NULL: EQU 0 ; Null terminator	
11	4				; Other code follows	
12	5				; - - -	

LONGEVEN (Forcing Long-Word Alignment): The next instruction will be forced to the next long-word address relative to the start of the section.

```
LONGEVEN
```

Repetitive Assembly

The Metrowerks HC12 assembler can generate multiple lines of code from one line of input code. This is useful when defining a block of memory constants that have some algorithmic relationship that can be defined at assembly time. Repetitive assembly can only be done if the assembler option `-Compat=b` is used. By default this is turned off.

FOR (Repeat Assembly Block):

ENDFOR (End of FOR Block): See Example 4-18.

Example 4-18 FOR Assembly Block

```

; for.asm
; Define a lookup table for a sawtooth
; wave whose output ranges from 0 to 24 in steps
; of 4, i.e. 0, 4, 8, . . .
; The assembly option -Compat=b must be used
    FOR label = 0 TO 6
        DC.B label*4
    ENDFOR

```

The following assembly code is generated:

Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj. code	Source line
1	1			; for.asm
2	2			; Define a lookup table for a sawtooth
3	3			; wave whose output ranges from 0 to 24 in steps
4	4			; of 4, i.e. 0, 4, 8, . . .
5	5			; The assembly option -Compat=b must be used
6	6			FOR label = 0 TO 6
7	7			DC.B label*4
8	8			ENDFOR
9	7	000000	00	DC.B label*4
10	8			ENDFOR
11	7	000001	04	DC.B label*4
12	8			ENDFOR
13	7	000002	08	DC.B label*4
14	8			ENDFOR
15	7	000003	0C	DC.B label*4
16	8			ENDFOR
17	7	000004	10	DC.B label*4
18	8			ENDFOR
19	7	000005	14	DC.B label*4
20	8			ENDFOR
21	7	000006	18	DC.B label*4
22	8			ENDFOR

Listing Control

The following directives let you control how your list file looks.

CLIST (List Conditional Assembly): The CLIST directive controls the listing of conditional assembly blocks.

```
CLIST    [ON | OFF]
```

When CLIST is on, the list file includes all directives and instructions in the conditional assembly block. Otherwise, only directives and instructions that generate code are listed.

LIST (Enable Listing) and NOLIST (Disable Listing):

```
LIST
```

```
NOLIST
```

LIST and NOLIST allow you to control portions of your code to be listing. You could use this to turn off listing of sections of the code that you are not interested in looking at, say for debugging. See Example 4-19.

Example 4-19 LIST and NOLIST

```

; list.asm
; Illustration of the LIST and NOLIST directives.
;
; This "code" shows in the listing.
    NOLIST
; While this line will not show.
    LIST
; Now listing is turned back on.
This source generates this assembly listing:
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003

Abs. Rel. Loc   Obj. code Source line
-----
1      1      1      ; list.asm
2      2      2      ; Illustration of the LIST and NOLIST directives.
3      3      3      ;
4      4      4      ; This "code" shows in the listing.
8      8      8      ; Now listing is turned back on.

```

LLEN (Set Line Length): Set the number of characters from the source file that are included on the listing line to <n>, which may be in the range of 0 to 132. Lines are truncated to this length.

```
LLEN <n>
```

MLIST (List Macro Expansions): MLIST is similar to CLIST in that it controls the listing of macro expansions in the list file. When ON (the default), macro expansions are included in the list file.

```
MSLIT [ON | OFF]
```

NOPAGE (Disable Paging): Program lines are listed continuously without headings or top or bottom margins.

```
NOPAGE
```

PAGE (Insert Page Break): Insert page break in the list file.

```
PAGE
```

PLEN (Set Page Length): Set the list page length to <n> lines, where <n> may range from 10 to 10000. The default page length is 65 lines.

```
PLEN <n>
```

SPC (Insert Blank Lines): Insert <count> blank lines in the list file, where <count> may range from 0 to 65.

```
SPC <count>
```

TABS (Set Tab Length): Sets the tab length to <n> spaces where <n> may be 0 to 128. The default is eight.

```
TABS <n>
```

TITLE (Provide Listing Title): The title given will be printed at the head of every page in the listing. This directive must be the first source code line and consists of a string of characters enclosed in quotes (").

```
TITLE "This is a title"
```

Macros

The following directives are associated with the macro assembler features.

MACRO (Begin Macro Definition): The <label> is name by which the macro is called to invoke it.

```
<label> MACRO
```

ENDM (End Macro Definition):

```
ENDM
```

MEXIT (Terminate Macro Expansion): The *MEXIT* directive allows a macro expansion to be terminated before the ENDM directive. This is usually done with conditional assembly within the macro.

```
MEXIT
```

Macro Assembler Operation

A macro assembler is one in which frequently used assembly instructions can be collected into a single statement. It makes the assembler more like a high-level language. For example, the problem might require a short code sequence to divide A by four

```
asra      ; Divide A by 2
asra      ; Divide by 2 again
```

in many parts of your program. This code is too short to be written as a subroutine so a macro is appropriate. Macros are often used for short segments of code, say fewer than ten assembler statements. There are three stages of using a macro. The first is the *macro definition*, and a typical definition is shown in Example 4-20. The assembler directives *MACRO* and *ENDM* encapsulate the code to be substituted when the macro is invoked. The macro definition may include any code or directive except for another macro definition. It may include a previously defined macro as well. The label *Delay_100* is used in the second stage - the *macro invocation*. It is written in the source program where the lines of code would normally be placed. The third stage, *macro expansion*, occurs when the assembler encounters the macro name in the source code. The macro name is expanded into the full code that was defined in the definition stage.

The syntax for macro invocation in the source file where you want to use it is

```
[<label>:] <macro_name> [<argument_list>]
```

This is *calling* a macro, much like calling a subroutine, except for each instance of the macro call the code is expanded and included in the assembly code.

Example 4-20 shows the macro definition, invocation and expansion. Relative (*Rel*) lines 6 – 13 show the macro definition bounded by the *MACRO* and *ENDM* directives. The first macro invocation occurs at *Rel* line 16 and the second at 19. The macro expansions are shown in *Abs* lines 17 – 22 and 26 – 31. Note that the relative line numbers show the line numbers from the macro definition and are denoted by the "m".

Example 4-20 Macro Definition

Abs.	Rel.	Loc	Obj. code	Source line
1	1			; macro.asm
2	2			;
3	3			; Here is the macro definition
4	4			;
5	5		0000 00C7	MU100: EQU 199 ; Delay loop counter
6	6			Delay_100 MACRO
7	7			; Macro to delay approximately 100 microseconds
8	8			psha ; Save the A reg
9	9			ldaa #MU100
10	10			\@loop: decb ; Label gets automatic number
11	11			bne \@loop; Delay until A=0
12	12			pula ; Restore A reg
13	13			ENDM
14	14			;
15	15			; Now Invoke the macro
16	16			Delay_100
17	7m			; Macro to delay approximately 100 microseconds
18	8m000000	36		psha ; Save the A reg
19	9m000001	86C7		ldaa #MU100
20	10m000003	43		_00001loop: decb ; Label gets automatic number
21	11m000004	26FD		bne _00001loop; Delay until A=0
22	12m000006	32		pula ; Restore A reg
23	17			;
24	18			; Do it again to illustrate the label change
25	19			Delay_100
26	7m			; Macro to delay approximately 100 microseconds
27	8m000007	36		psha ; Save the A reg
28	9m000008	86C7		ldaa #MU100
29	10m00000A	43		_00002loop: decb ; Label gets automatic number
30	11m00000B	26FD		bne _00002loop; Delay until A=0
31	12m00000D	32		pula ; Restore A reg

Labels in a Macro

Multiple occurrences of a label in a program are normally not allowed, and for this reason the programmer can direct the assembler to create unique labels for each macro invocation. These assembler-generated labels are in the form `_nnnnn` where `nnnnn` is a five digit number. The programmer specifies this to occur by specifying `\@` in the label field within the macro body. You can see how this works in Example 4-20 where *Abs* lines 20 and 29 have unique labels generated by the assembler.

Macro Parameters

Substitutable parameters can be used in the source statement when the macro is called.

These allow you to use a macro in different applications. Up to 36 parameters may be specified in the macro definition by a backslash character (\n) where 'n' is the nth parameter and may be the digits 0 – 9 or an uppercase letter A- Z. When the macro is called, arguments from the argument list are substituted into the body of the macro as literal string substitutions. Example 4-21 shows a macro definition and call using a parameter to control how many times the A register is shifted left. The parameter, `Num_shift`, appears in the macro definition to remind you how it is used and to be helpful in documentation. Multiple parameters are separated by commas and a null argument may be passed by two commas (,,).

Parameter zero (0) corresponds to a size argument that may follow the macro name, separated by a period (.). Another useful feature is macro argument grouping. If you wish to pass text with commas as a macro parameter, you may group these using a special syntax. The argument group is delimited by a `[?` as a prefix and `?]?` as the suffix. See Example 4-22

Example 4-21 Macro parameters

Metrowerks HC12-Assembler

(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj. code	Source line
1	1			; macro_param.asm
2	2			; Macro definition for a variable arithmetic
3	3			; shift left of accumulator A
4	4			alsan MACRO Num_shift
5	5			; Shift accumulator A left Num_shift bits
6	6			; where Num_shift is a parameter in
7	7			; the macro call.
8	8			; Save B to set up a loop counter
9	9			pushb ; Save B on the stack.
10	10			ldab #\1 ; Get Num_shift
11	11			\@loop: asla ; Shift A
12	12			dbne b,\@loop
13	13			pulb ; Restore the B
14	14			ENDM
15	15			
16	16			; The macro call is with a parameter
17	17			alsan 4 ; Shift A 4 bits left
18	5m			; Shift accumulator A left Num_shift bits
19	6m			; where Num_shift is a parameter in
20	7m			; the macro call.
21	8m			; Save B to set up a loop counter
22	9m000000	37		pushb ; Save B on the stack.
23	10m000001	C604		ldab #4 ; Get Num_shift
24	11m000003	48		_00001loop: asla ; Shift A
25	12m000004	0431 FC		dbne b,_00001loop
26	13m000007	33		pulb ; Restore the B
27	18			;

Example 4-22 Macro Parameter \0

Metrowerks HC12-Assembler

(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj. code	Source line
1	1			; macro_param_0.asm
2	2			; Illustration of macro parameter \0 and
3	3			; macro argument grouping
4	4			;
5	5			; Define data macro
6	6			define: MACRO
7	7			
8	8			DC.\0 \1,\2
9	9			ENDM
10	10			
11	11			; Use the define macro to define bytes
12	12			define.B \$10, 'A'
13		7m		
14		8m000000	1041	DC.B \$10,'A'
15	13			; Use the define macro to define words
16	14			define.W \$10,\$0d0a
17		7m		
18		8m000002	0010 0D0A	DC.W \$10,\$0d0a
19	15			; Use the define macro with argument grouping
20	16			define.B \$10,['A','B','C?']
21		7m		
22		8m000006	1041 4243	DC.B \$10,'A','B','C'
23	17			; You can use argument grouping with strings
24	18			define.B \$10,['This is a string?']
25		7m		
26		8m00000A	1054 6869	DC.B \$10,'This is a string'
		00000E	7320 6973	
		000012	2061 2073	
		000016	7472 696E	
		00001A	67	

Macros and Subroutines

Macros and subroutines have similar properties.

! Both allow the programmer to reuse segments of code. However, each time a macro is invoked, the assembler expands the macro and the code appears "in line." A subroutine code is included only once. Thus, macro expansions make the program larger.

! The subroutine requires a call or jump-to-subroutine and the macro does not. This means that the subroutine is a little slower to execute than the macro and the subroutine call uses stack space temporarily.

! Both macros and subroutines allow changes to be made in one place (the macro definition or the subroutine).

! Macros and subroutines make the program easier to read by hiding details of the program. Usually, when reading a program, you do not need the details of how it is doing something, just an indication of what it is doing.

Conditional Assembly

You may have seen the IF-THEN-ELSE structure in a high-level language. We know the code for the THEN part is executed if the conditional is true, and the ELSE part if the conditional is false. A conditional assembly is very similar but only the appropriate segment of code is included in the assembled program. Note that this *does not* produce code that is an executable IF-THEN-ELSE structure. We will see how to write structured code in Chapter 7.

The conditional assembly feature allows you to write code that can be customized at assembly time. Example 4-23 shows how to choose a set of equates for one of two different versions of the software. Conditional assembly may be invoked with the following directives:

IF (Conditional Assembly): Start of conditional assembly block.

```
IF    <condition>
      [assembly language statements if true]
[ELSE]
      [assembly language statements if false]
ENDIF
```

If <condition> is true, the [assembly language statements if true] code are assembled. Assembly continues until the ELSE or EXITM directive is encountered. Nesting of conditional blocks is allowed limited only by the amount of memory at assembly time.

The <condition> is a Boolean and its syntax is:

```
<condition> := <expression> <relation> <expression>
<relation> := "=" | "!=" | ">=" | ">" | "<=" | "<" | "<>"
```

The <expression> must be able to be evaluated at assembly time.

IFcc (Conditional Assembly): An alternative conditional assembly syntax may be used.

```
IF    <condition>
      [assembly language statements if true]
[ELSE]
      [assembly language statements if false]
ENDIF
```

Table 4-6 shows the available conditional types.

Table 4-6 Conditional assembly types.

IFcc	Condition	Meaning
IFeq	<expression>	If <expression> == 0
IFne	<expression>	If <expression> != 0
IfFt	<expression>	If <expression> < 0
IFle	<expression>	If <expression> <= 0
IFgt	<expression>	If <expression> > 0
IFge	<expression>	If <expression> >= 0
IFc	<string1>,<string2>	If <string1> == <string2>
IFnc	string1,<string2>	If <string1> != <string2>
IFdef	<label>	If <label> was defined
IFndef	<label>	If <label> was not define

Example 4-23 Conditional Assembly

The conditional assembly code is the following:

```

; ifelse.asm
TRUE: EQU 1
FALSE: EQU 0
; Define parameters for each version
Param1: EQU $76 ; Use in version 1
Param2: EQU $77 ; Use in version 2
; Set the version number for this software
Ver1: EQU TRUE

; Here is the conditional assembly:
IF Ver1 = TRUE
    ldaa #Param1
ELSE
    ldaa #Param2
ENDIF

;
; You can also use the following form:
IFNE Ver1 ; If Ver1 not equal to zero
    ldaa #Param1
ELSE
    ldaa #Param2
ENDIF

```

The assembler generates the following code:

Metrowerks HC12-Assembler

(c) COPYRIGHT METROWERKS 1987-2003

Abs.	Rel.	Loc	Obj.	code	Source line
----	----	-----	-----	-----	-----
1	1				; ifelse.asm
2	2		0000 0001		TRUE: EQU 1
3	3		0000 0000		FALSE: EQU 0
4	4				; Define parameters for each version
5	5		0000 0076		Param1: EQU \$76 ; Use in version 1
6	6		0000 0077		Param2: EQU \$77 ; Use in version 2
7	7				; Set the version number for this software
8	8		0000 0001		Ver1: EQU TRUE
9	9				
10	10				; Here is the conditional assembly:
11	11		0000 0001		IF Ver1 = TRUE
12	12	0000000	8676		ldaa #Param1
13	13				ELSE
15	15				ENDIF
16	16				;
17	17				; You can also use the following form:
18	18		0000 0001		IFNE Ver1 ; If Ver1 not equal to zero
19	19	0000002	8676		ldaa #Param1
20	20				ELSE
22	22				ENDIF

4.6 Assembler Files

The assembler produces a variety of output files including the *assembler listing*, and the *debug listing* in addition to object output files. The object files are, one way or another, destined to be loaded into the microcontrollers memory while the listing files are used for documentation and debugging. Table 4-7 shows the extension used and the location for all assembler input and output file.

Assembler Listing

The assembler listing file contains information about the generated code and is generated when the `-L` assembler option is activated. If an assembler error occurs, no listing file is generated. The file has the format shown in Example 4-1 and contains the following fields of information. The assembler list file has the extension *.lst*.

Abs: This column contains the line number for each instruction. It is *absolute* in the sense that it enumerates all included files and where all macro calls have been expanded.

Rel: The *relative* line number is the line number in the source file. For included files, the relative line number is the line number in the included file and for macro expansion it is the line number of the instruction in the macro definition. In each of these cases, an 'i' and 'm' suffix is appended to the relative line number.

Loc: The *loc* column contains the address of the instruction. In absolute sections of your program an 'a' precedes the address. In relocatable sections, the address is the offset from the beginning of the section. The address is given in hexadecimal with up to six digits.

Obj. code: This column contains the hexadecimal code for each instruction. If, in relocatable section, parts of the code that have not been resolved (to be done by the linker) are displayed as 'x'.

Source Line: Each line in the source file is repeated in this column.

Debug Listing

The debug listing file, created with the extension *dbg*, contains debugging information for the Metrowerks True-Time Simulator.

Object Files

Object files always have the extension *.o* and contain the target code as well as some debugging information.

Absolute Files

Absolute files may be created when an application is encoded in a single module and all sections are absolute sections. The absolute file has the extension *.abs*.

Freescale S Files

When an application is encoded in a single module with all absolute sections, you may generate an ELF absolute file instead of an object file. A Freescale S record file is generated at the same time that can be burned into an EPROM or downloaded to a

development board equipped with a debugging monitor such as D-Bug12. The S record file will have an extension `.sx` or `.s1`, `.s2`, or `.s3`.

Table 4-7 Assembler Files

Extension	File	Where Found
<code>.o</code>	Binary object	Source file directory or OBJPATH
<code>.dbg</code>	Debug listing	Source file directory or OBJPATH
<code>.abs</code>	Absolute file	Source file directory or ABSPATH
<code>.asm</code>	Source file	Project directory or GENPATH
<code>.inc</code>	Include file	Project directory or GENPATH
<code>.sx</code>	Freescaler S record	Source file directory or ABSPATH
<code>.s1</code>	Freescaler S record if SRECORD = 1	Source file directory or ABSPATH
<code>.s2</code>	Freescaler S record if SRECORD = 2	Source file directory or ABSPATH
<code>.s3</code>	Freescaler S record if SRECORD = 3	Source file directory or ABSPATH
<code>Err.txt</code>	Error listing	Current directory

4.7 Remaining Questions

4.8 Conclusion and Chapter Summary Points

4.9 Bibliography and Further Reading

Cady, F. M., *Microcontrollers and Microprocessors*, Oxford University Press, New York, 1997.

Motorola HC12 Assembler, Metrowerks Corporation, Austin, TX, 2003.

4.10 Problems

Table of Contents

<i>Chapter 4</i>	1
4.1 Assembly Language Example.....	1
4.2 Metrowerks HC12 Assembler.....	3
4.3 Assembler Source Code Fields	3
Label Field	3
Opcode or Operation Field	4
Operand Field	5
Comment Field.....	9
4.4 Assembler Control	10
4.5 Assembler Directives	11
Section Definition.....	11
Constant Definition.....	14
Reserving or Allocating Memory Locations	15
Defining Constants in Memory.....	16
Export or Import Global Symbols.....	18
Assembly Control	19
Repetitive Assembly	20
Listing Control.....	21
Macros.....	23
Macro Assembler Operation.....	23
Labels in a Macro.....	24
Macro Parameters	24
Macros and Subroutines.....	26
Conditional Assembly.....	27
4.6 Assembler Files.....	30
Assembler Listing	30
Debug Listing.....	30
Object Files	30
Absolute Files	30
Freescale S Files.....	30
4.7 Remaining Questions	31
4.8 Conclusion and Chapter Summary Points	31
4.9 Bibliography and Further Reading.....	31
4.10 Problems	31

Chapter Listings:
Revision Chapters

1. Introduction
2. General Principles of Microcontrollers
3. Introduction to the M68HC(S)12 Hardware
4. An Assembler Program – CodeWarrior
5. A Linker Program
6. The M68HC12 Instruction Set
7. Assembly Language Programs for the M68HCS12
8. Simulating M68HCS12 Programs
9. Debugging M68HCS12 Programs
10. Program Development Using C
11. M68HCS12 Parallel I/O
12. M68HCS12 Interrupts
13. M68HCS12 Memories
14. M68HCS12 Timer
15. M68HCS12 Serial I/O – SCI, SPI
16. M68HCS12 Serial I/O – CAN, IIC
17. M68HCS12 Analog Input
18. Single-Chip Microcomputer Interfacing Techniques
19. Fuzzy Logic
20. Debugging Systems
21. Advanced M68HCS12 Hardware
22. Appendix A – D-Bug12 Monitor
23. Appendix B – Debugging Systems Pod Design
24. Appendix C – Notations and References