CodeWarrior®

SH Assembler Reference

metrowerks®

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 990224 rw

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite 100 Austin TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	go `Metrowerks`

Table of Contents

1

Introduction

This manual describes the syntax for the Metrowerks Assemblers, and uses SH assembler for the examples.

Overview of the Assembler Manual

This manual describes Metrowerks Assemblers, a collection of assemblers for several different processors. This manual describes the syntax for statements, macros, and directives.

This manual assumes you are already familiar with assembler and the processor you're writing code for.

The chapters in this manual include:

- Assembler Syntax Overview
- Using Macros Overview
- Using Directives Overview
- Assembler Settings Overview

For more information on the instruction mnemonics and register names for a particular processor, see the books described in "Where to Learn More."

What Are the Metrowerks Assemblers

The Metrowerks Assemblers is a collection of assemblers for several different processors. They share an identical syntax for statements, directives, and macros. They differ only in the instruction mnemonics and register names that are used for each processor.

The *SH Assembler* supports all instructions for the SH-4 processor.

NOTE: The assembler commands described in this manual refer to the Metrowerks Assembler itself and should not be confused with the inline assembler included in the Metrowerks C/C++ compilers. See the *Dreamcast Targeting Manual* and the *C Compilers Reference* for additional information on using the C/C++ inline assembler.

Conventions Used in This Manual

This manual includes syntax examples that describe how to use certain statements, Table 1.1 describes how to interpret these statements.

Table 1.1 Understanding Syntax Examples

If the text looks like…	Then…
`literal`	Include it in your statement exactly as it's printed.
metasymbol	Replace the symbol with an appropriate value. The text after the syntax example describes what the appropriate values are.
`a | b | c`	Use one and only one of the symbols in the statement: either a, b, or c.
`[a]`	Include this symbol only if necessary. The text after the syntax example describes when to include it.

Where to Learn More

The assembler uses the standard mnemonics and register names as defined in the following documents:

- SH: *SH-4 Hardware Manual*, Hitachi Ltd.

2
Assembler Syntax

The chapter describes the sytax rules required to write a source file for the Metrowerks Assemblers.

Assembler Syntax Overview

This chapter explains how to write a source file for the Metrowerks Assemblers. You should already be familiar with Assemblers and with the machine operations for the processor you're writing for. This chapter does not describe the instructions for the processors. For more information, see "Where to Learn More."

This chapter covers the following topics:

- Statement Syntax
- Symbol Syntax
- Symbol Scope
- Constant Syntax
- Expression Syntax
- Forward Equate Syntax
- Data Alignment

Statement Syntax

An assembly language statement may be one these types:

- Instruction statement
- Directive statement
- Macro statement

A statement can be no more than 1000 characters long. You cannot split a statement across multiple source lines, and you cannot put more than one statement on a source line.

A statement has the following syntax:

Listing 2.1 Statement syntax

```
[label] operation [operands] [comment]
```

Here are the parts of an assembler statement:

Label By default, a *label* ends in a colon (:) and can begin in any column. If you're porting existing code that doesn't follow this convention, turn off the **Labels must end with** ':' option. With the option turned off, a symbol is a label if it starts in column 1 *or* if it ends with a colon (:).

For more information on labels, see "Symbol Syntax."

Operation The *operation* is a name for one of the following:

- A machine operation. To find out which machine instructions are allowed for a particular chip, see "Where to Learn More."
- A macro call. For more information on macros, see "Using Macros Overview."
- An assembler directive. For a list of all the assembler directives, see "Using Directives Overview."

Instruction, directive, and macro names are case insensitive. For example `MOV`, `Mov`, and `mov` all name the same instruction.

Operands The *operands* specify the data that the operation uses. The type of the operation determines how many operands are required, if any. To separate operands, use a comma (,).

Comments Comments are text that the Metrowerks Assembler ignores and are useful for documenting your code. The Metrowerks Assembler ignores any text between a semicolon (;) and the end of the line. To help you port existing code, the Metrowerks Assembler treats the following text as a comment:

- In all current assemblers, it ignores any text between an asterisk (*) at the beginning of the line and the end of the line. Note that the asterisk must be the first character in the line. It has other meanings when it occurs elsewhere in a line. Also note that some future assemblers may use an asterisk at the beginning of the line for another purpose.
- In all current assemblers, if you turn off the **Allow space in operand field** option, it ignores any text between a space character in the operand field and the end of the line. However, some future assemblers may use this for another purpose.

Symbol Syntax

A *symbol* is a combination of characters that represents a value, such as an address, numeric constant, string constant, or character constant. The two types of symbols are labels and equates. A *label* is a symbol that represents an address. An *equate* is a symbol that represents any value and that you create with a `.equ` or `.set` directive.

A symbol's name has unlimited length and can contain the following:

- The first character must be one of these:
  - If it's not a local label, a-z, A-Z, a period (.), a question mark (?), or underscore (_).
  - If it is a local label, at-sign (@).
- Each remaining character must be one of a-z, A-Z, numerals 0-9, underscore (_), question mark (?), dollar sign ($), or period (.).

The **Case sensitive identifiers** option lets you choose whether symbols are case-sensitive. If the option is on, symbols are case sensitive, so `SYM1`, `sym1`, and `Sym1` are three different symbols, for example. If the option is off, symbols are *not* case-sensitive, so `SYM1`, `sym1`, and `Sym1` are the same symbol, for example. By default, this option is on.

To refer to the the program counter use one of these characters: period (.), dollar sign ($), or asterisk (*).

Symbol Scope

In general, a symbol has file-wide scope: you can access it anywhere within the file it is defined and only within the file it is defined.

A label can also have local scope: you can access it forwards and backwards until a non-local label is encountered. To create a local label, begin its name with an at-sign (@).

An equate can also have global scope: you can access it from other files. To create an global equate, use the `.global` or `.public` directives.

In this section we discuss:

- Local Labels
- Global Equates

NOTE: An equate cannot have local scope. A local label cannot have global scope while a normal label can.

Local Labels

Labels whose names begin with the at-sign (@) character are local. The scope of a local label extends forwards and backwards until a non-local label is encountered. A forward equate (described in "Forward Equate Syntax") does not end the scope.

Lines generated by macro expansion have their own name scope for local labels:

- A non-local label in an expanded macro does not end the scope of locals in the unexpanded source
- The scope of local labels defined in macros does not extend outside the macro.

The following example illustrates the scope of local labels in macros.

Listing 2.2 The scope of local labels in a macro

```
MAKEPOS   .MACRO
     tst     #1, r0
     bra     @SKIP
     neg     r0
@SKIP:  ;Scope of this label is within the macro
    .ENDM
START:
     mov.l   COUNT, r1
     cmp/eq  #1, r1
     bra     @SKIP
     MAKEPOS
@SKIP:    ;Scope of this label is START to END
          ;excluding lines arising from
          ;macro expansion
      add    #1, r0
END:  rts
```

In this example, the `@SKIP` label defined in the macro does not conflict with the `@SKIP` label defined in the main body of code.

Within a macro, the Metrowerks Assembler replaces the unique label symbol (\@) with a unique label name which you can use as a local label or a forward equate. For more information on unique labels see “Creating Unique Labels.” For more information on forward equates, see “Forward Equate Syntax.”

NOTE: You cannot export local labels and that local labels do not appear in debugging tables.

Global Equates

An equate can also have global scope: you can access it from other files. To declare an equate to have global scope, use the `.global` directive, like this:

Listing 2.3 Declaring a global equate

```
CONST      .set     256
           .global CONST
```

To access the equate from another file, use the `.extern` directive, like this:

Listing 2.4 Importing a global equate

```
.extern CONST
add     CONST, r1
```

Alternatively, you can also use the `.public` directive to both declare and import a global equate. If the specified equate is already defined, it's declared global. If the specified equate isn't defined, it's imported.

The name of your symbol may change when you import a global equate from C or C++. View the disassembly of the source file and look in the symbol table for the hashed symbol name. For example, you may have a source file in which you define the following global variable:

```
unsigned long  this_long = 0x12345678;
```

When you view the disassembly of the source file, you will see that the name has changed. In this example, an underscore character '_' was added to the name.

```
                 *** SYMBOL TABLE (.symtab) ***
no  value       size       bind   type    other  shndx  name
13  0x00000000  0x00000004 GLOBAL OBJECT  0x00  .data   _this_long
```

Relocatable Labels

The Metrowerks Assembler assumes a flat 32-bit memory space. You can specify the relocation of a 32-bit label with the following expressions.

NOTE: Some expressions are not allowed in all assemblers.

Table 2.1 Relocatable label expressions

This…	Represents this
label	The offset from the label to the base of its section, relocated by the section base address. It’s also the PC-relative target of a branch or call. It is a 32-bit address.
label`@l`	The low 16-bits of the symbol's relocated address
label`@h`	The high 16-bits of this address. You can OR this with *label*`@l` to produce the full 32-bit relocated address
label`@ha`	The adjusted high 16-bits of this address You can add this to *label*`@l` to produce the full 32-bit relocated address
label`@sdax`	For labels in a small data section, the offset from the base of the small data section to the label. This syntax is not allowed for labels in other sections.
label`@got`	For chips with a global offset table, the offset from the base of the global offset table to the 32-bit entry for label

Constant Syntax

The Metrowerks Assembler recognizes three kinds of constants:

- Integer Constants
- Floating Point Constants
- Character Constants

The syntax for each type of constant is the same in all assemblers.

Integer Constants

This table lists the preferred notation for integer contants. This notation works for all Metrowerks Assemblers.

Table 2.2 Preferred integer constant notation

For numbers of this type…	Use…
Decimal	A string of decimal digits, such as `12345678`.
Hexadecimal	A dollar sign ($) followed by a string of hexadecimal digits, such as `$deadbeef`.
Binary	A percent sign (%) followed by a string of binary digits, such as `%01010001`.

To help you port existing code, the current assemblers also support the notation in the following table. However, some future assemblers may use this notation for other purposes:

Table 2.3 Alternate integer constant notation

For numbers of this type…	Use…
Hexadecimal	`0x` followed by a string of hexadecimal digits, such as `0xdeadbeef`.
Hexadecimal	`0` followed by a string of hexadecimal digits, such as `0deadbeef`, and ending with an `h`, such as `0deadbeefh`.
Decimal	A string of decimal digits followed by `d`, such as `12345678d`.
Binary	A string of binary digits followed by a `b`, such as `01010001b`.

Note that the Metrowerks assemblers store and manipulate integer constants using 32-bit signed arithmetic.

Floating Point Constants

You can specify floating point constants in either hexadecimal or decimal format. A floating point constant in decimal format must contain either a decimal point or an exponent, e.g. `1E-10` or `1.0`.

You can use floating point constants only in data generation directives like `.float` and `.double`, or in floating point instructions. You cannot use them in expressions.

Character Constants

A character constant must be enclosed in single quotes, and can be up to 4-characters wide depending on the context; for example, `'A'`, `'ABC'`, and `'TEXT'.`

To specify a single quote (`'`) within a character constant, use two single quote characters; for example, `'IT''S'`. A character constant can also contain any of these escape sequences.

Table 2.4 Escape sequences

Sequence	Description
`\b`	Backspace
`\n`	Line feed (ASCII character 10)
`\r`	Return (ASCII character 13)
`\t`	Tab
`\"`	Double quote
`\\`	Backslash
`\nnn`	Octal value of `\nnn`

A character constant is zero-extended to 32 bits during computation. You can use a character constant anywhere you can use an integer constant.

Expression Syntax

The Metrowerks Assemblers evaluate expressions using 32-bit signed arithmetic. They do not check for arithmetic overflow.

Since there is no common set of operators in the existing assemblers for different processors, the Metrowerks assemblers use an expression syntax similar to the one for the C language. Expressions use the C language arithmetic rules for such things as parentheses and associativity, and they use the same operators.

All the Metrowerks Assemblers support the operators listed in these tables:

Table 2.5 Binary operators

Operator	Description
+	add
-	subtract
*	multiply
/	divide
%	modulo
||	logical OR
&&	logical AND
|	bitwise OR
&	bitwise AND
^	bitwise XOR
<<	shift left
>>	shift right (zeros are shifted into high order bits)
==	equal to
!=	not equal to
<=	less than or equal to
>=	greater than or equal to
<	less than
>	greater than

Table 2.6 Unary operators

Operator	Description
+	unary plus
-	unary minus
~	unary bitwise complement

All the current Metrowerks Assemblers also allow the operations listed in Table 2.7. However, some future assemblers may reserve these operators for other purposes.

Table 2.7 Alternate operators

Operator	Description
<>	not equal to
//	modulo
!	logical OR
!!	logical XOR

The operators have the following precedence, with the highest priority first:

1. unary + - ~
2. * / %
3. binary + -
4. << >>
5. < <= > >=
6. == !=
7. &
8. ^
9. |
10. &&
11. ||

Forward Equate Syntax

The Metrowerks Assemblers allow *forward equates*: This lets you refer to a symbol in a file before it is defined. When an assembler comes across an expression it cannot resolve because the expression references a symbol whose value is not known, the assembler retains the expression and marks it as unresolved. After the assembler reads the whole file, it re-evaluates unresolved expressions and, if necessary, repeatedly re-evaluates them until it resolves them all or it cannot resolve them any further. If the assembler cannot resolve an expression, it raises an error.

However, the assembler must be able to immediately resolve any expression whose value affects the location counter.

NOTE: Note that if the assembler can make a reasonable assumption about the location counter, the expression is allowed. For example, in a forward branch instruction for a 68K processor, you can specify a default assumption of 8, 16, or 32 bits.

Thus, the code in Listing 2.5 is allowed.

Listing 2.5 Valid forward equate

```
              .long  alloc_size
alloc_size    .set   rec_size + 4
                     ; a valid forward equate on next line
rec_size      .set   table_start-table_end
   ;...
table_start:
   ; ...
table_end:
```

However, the code in the following example is not allowed. The assembler cannot immediately resolve the expression in the `.space` directive, so the effect on the location counter is unknown.

Listing 2.6 Invalid forward equate

```
                   ;invalid forward equate on next line
rec_size   .set   table_start-table_end
           .space rec_size
   ; ...
table_start:
   ; ...
table_end:
```

Data Alignment

By default, all data is aligned on a natural boundary for the data size and for the target processor family. You may turn off alignment with the `alignment` argument to the `.option` directive, described in "option."

An assembler does not align data automatically in the `.debug` section. For more information on the `.debug` section, see "Debugging Directives."

3

Using Macros

This chapter describes how to define and use macros.

Using Macros Overview

The Metrowerks Assemblers let you use the same macro language for any of the target processors. Note that the macro language is broadly similar to Hitachi assembler syntax with some extensions.

This chapter describes the following:

- Defining Macros
- Invoking Macros

Defining Macros

This section describes how to define a macro. It tells you about the following:

- Macro Definition Syntax
- Using Macro Arguments
- Referring to the Number of Arguments
- Creating Unique Labels

Macro Definition Syntax

A *macro definition* is a sequence of assembly statements that defines the name of a macro, the format of its call, and the assembly statements to process when it's invoked. It looks like this:

Listing 3.1 A macro definition

```
name:    .macro  [ param1,] [ param2 ]. . .
         ; macro body
         .endm
```

The *name* is a label used to invoke the macro. You can include an optional list of parameters, like *param1* and *param2*, which are operands passed to the macro and are used in the macro body. The *macro body* consists of assembler statements that are substituted for a macro call when you invoke the macro.

The macro definition must end with `.endm`. If you want to stop macro processing before `.endm` is reached (for example, the macro may contain conditional assembly), use `.mexit`.

Using Macro Arguments

You can refer to parameters directly by name. Here is the `setup` macro, which moves an integer into `d0` and branches to the label `_final_setup`:

Listing 3.2 The setup macro

```
setup:       .macro name
             mov.l  #name, r0
             bsr    _final_setup
             .endm
```

If you invoke it like this:

Listing 3.3 Calling setup

```
              setup  'VECT'
```

It’s expanded like this:

Listing 3.4 Expanded setup

```
           MOV.L  #'VECT', R0
           BSR    _set_it_up_
```

When you refer to named macro parameters in the macro body, you can precede or follow the macro parameter with &&. This lets you embed the parameter in a string. For example, here is the `smallnum` macro, which creates a small float by appending the string `E-50` to the macro's argument:

Listing 3.5 The smallnum macro

```
smallnum:    .macro    mantissa
             .float    mantissa&&E-50
             .endm
```

If you invoke it like this:

Listing 3.6 Invoking smallnum

```
              smallnum  10
```

It's expanded like this:

Listing 3.7 Expanding smallnum

```
              .float     10E-50
```

Creating Unique Labels

You can generate unique labels within a macro with the symbol `\@`. Each time you invoke the macro, the assembler generates a unique symbol of the form `??`*nnnn*, such as `??0001`, or `??0002` each time the macro is called.

Also, a local label, which is any label that begins with @, has a scope which is restricted to only the expansion of the macro. For more information, see “Symbol Scope.”

Unique labels and symbols (those that use \@) are referred to in your code with the same methods used for regular labels and symbols. The \@ sequence gets replaced by a unique string which is incremented each time the macro is invoked.

Listing 3.8 Unique label macro

```
my_macro: .macro
          foo\@ = my_count
my_count  .set my_count + 1
          add  fred\@, r1
          bra  label\@
          add  r1, r2
label\@:
          nop
.endm
```

If the macro in Listing 3.8 is called twice (with `my_count` initialized to 0), it gets assembled into something like Listing 3.9.

Listing 3.9 Unique label assembler output

```
0x00000000:     foo??0000   =     my_count
0x00000001:     my_count    .set  my_count + 1
0x00000008:                 add   fred??0000, r1
0x0000000c:                 bra  label??0000
0x00000010:                 add   r1, r2
0x00000014:     label??0000
0x00000014:                 nop
0x00000000:                 my_macro
0x00000000:     fred??0001  =     my_count
0x00000001:     my_count    .set  my_count + 1
0x00000008:                 add   fred??0001, r1
0x0000000c:                 bra   label??0001
0x00000010:                 add   r1, r2
0x00000014:     label??0001
0x00000014:                 nop
0x00000000:
```

Referring to the Number of Arguments

To refer to the number of non-null arguments passed to a macro, use the special symbol `narg`. You can use it only during macro expansion.

Invoking Macros

To invoke a macro, simply use its name in your assembler listing.

When invoking a macro, you must separate parameters with commas. To pass a parameter that includes a comma, enclose the parameter in angle brackets. For example, here is a statement that calls a macro named `moveit`, that expands to the `mov.l` instruction

Listing 3.10 Invoking moveit with an argument that contains commas

```
              moveit.l   <@(1020,pc)>, r15
```

4

Using Directives

This chapter describes the directives that are available in any Metrowerks Assembler.

Using Directives Overview

This chapter documents how to use assembler directives in a Metrowerks Assembler. Some directives are not available in every assembler. The directive's description notes which assemblers support the directive.

By default, directives must begin with a period (.). However if you turn off the **Directives begin with '.'** option in the Assembler settings panel, you can leave out the period.

The rest of this chapter lists the directives, arranged in these categories:

- Macro Directives
- Conditional Preprocessor Directives
- Section Control Directives
- Scope Control Directives
- Symbol Definition Directives
- Data Declaration Directives
- Assembler Control Directives
- Debugging Directives

Macro Directives

The following directives let you create macros. For more informations on macros, see “Using Macros Overview.”

- `macro`–begins a macro definition.
- `endm`–ends a macro definition.
- `mexit`–terminates a macro’s expansion before it reaches `endm`.

macro

```
label        .macro [ param1, param2 . . . ]
```

Begins the definition of a macro named *label*, with the specified parameters.

endm

```
.endm
```

Ends a macro definition.

mexit

```
.mexit
```

Ends the expansion of macro before it reaches `.endm`.

Conditional Preprocessor Directives

Conditional directives create a conditional assembly block. If you wrap some code with `.ifdef` and `.endif` you can control whether that code is included in compilation. This is useful for making several different builds that are slightly different.

You must use conditional directives together to form a complete block. The Metrowerks Assemblers also contain several variations of `.if` to make it easier to make blocks that test strings for equality, test whether a symbol is defined, and more. Here are the directives.

- `if`–begins conditional assembly and uses any Boolean expression.
- `ifdef`–begins conditional assembly and tests whether a symbol is defined.
- `ifndef`–begins conditional assembly and tests whether a symbol is *not* defined.
- `ifc`–begins conditional assembly and tests whether two strings are equal.
- `ifnc`–begins conditional assembly and tests whether two strings are *not* equal.
- `endif`–ends conditional assembly.
- `elseif`–marks another test to make, if the first test returned false.
- `elif`–marks another test to make, if the first test returned false. This is just like `elseif`.
- `else`–marks statements to execute if none of the tests succeeded.
- `ifeq ifne iflt ifle ifgt ifge`–are additional conditional assembly statements for backwards compatibility.

if

`.if` *bool-expr*

Specifies the beginning of conditional assembly, where *bool-expr* is a boolean expression. If *bool-expr* is true, the assembler processes the statements associated with the `.if` directive. If *bool-expr* is false, the assembler skips the statements associated with the `.if` directive.

Each `.if` directive must have a matching `.endif` directive.

NOTE: A boolean expression is a special type of arithmetic expressions. A boolean expression that evaluates to zero result is interpreted as false, and a boolean expression that evaluates to a nonzero result is interpreted as true. For more information on expressions, see "Expression Syntax."

ifdef

`.ifdef` *symbol*

Specifies the beginning of conditional assembly, where *symbol* is a the name of a symbol that has been defined. If name has been previously defined, the assembler processes the statements associated with the `.ifdef` directive. If name has *not* been previously defined, the assembler skips the statements associated with the `.ifdef` directive.

Each `.ifdef` directive must have a matching `.endif` directive.

ifndef

`.ifndef` *symbol*

Specifies the beginning of conditional assembly, where *symbol* is the name of a symbol that has *not* been defined. If name has *not* been previously defined, the assembler processes the statements associated with the `.ifndef` directive. If name has been previously defined, the assembler skips the statements associated with the `.ifndef` directive.

Each `.ifndef` directive must have a matching `.endif` directive.

ifc

`.ifc` *string1*`,` *string2*

Specifies the beginning of conditional assembly, where *string1* and *string2* are two strings that are equal. The comparison is case-sensitive. If the strings are equal, the assembler processes the statements associated with the `.ifc` directive. If the strings are *not* equal, the assembler skips the statements associated with the `.ifc` directive.

Each `.ifc` directive must have a matching `.endif` directive.

ifnc

`.ifnc` *string1*`,` *string2*

Specifies the beginning of conditional assembly, where *string1* and *string2* are two strings that are *not* equal. The comparison is case-sensitive. If the strings are *not* equal, the assembler processes the statements associated with the `.ifnc` directive. If the strings are equal, the assembler skips the statements associated with the `.ifnc` directive.

Each `.ifnc` directive must have a matching `.endif` directive.

endif

`.endif`

Marks the end of conditional assembly. Each type of `.if` directive must have a matching `.endif` directive.

elseif

`.elseif` *bool-expr*

Marks the beginning of conditional assembly statements to be processed if the Boolean expression for an `.if` directive and the preceding `.elseif` directives are false, but the *bool-expr* in this `.elseif` statement is true. An `.if` directive does not need an `.elseif` directive.

If the Boolean expression for an `.if` directive is false, the assembler skips the statements associated with the `.if` directive and evaluates the Boolean expression for the first `.elseif` directive. If that Boolean expression is true, the assembler processes the statements associated with that `.elseif` statement. Otherwise, it evaluates the Boolean expression in the next `.elseif` statement. The assembler continues evaluating the Boolean expressions in succeeding `.elseif` statement until it comes to a Boolean expression that evaluates to true. If none of the `.elseif` directives in the `.if-.endif` block have a Boolean expression that evaluates to true, the assembler processes the statements associated with the block's `.else` statement, if there is one.

elif

`.elif` *bool-expr*

This is the same as elseif.

else

```
.else
```

Marks the beginning of conditional assembly statements to be processed if the Boolean expression for an `.if` directive and its associated `.elseif` directives are false. An `.if` directive does not need an `.else` directive.

ifeq ifne iflt ifle ifgt ifge

```
.ifeq   ; if equal
.ifne   ; if not equal
.iflt   ; if less than
.ifle   ; if less than or equal
.ifgt   ; if greater than
.ifge   ; if greater than or equal
```

For compatibility with other assemblers, these directives are also supported.

Section Control Directives

These directives mark the different sections of an assembly file. All are available in all current Metrowerks Assemblers, but some future assemblers may not support all of them.

- `text`–specifies an executable code section.
- `data`–specifies an initialized read-write data section.
- `rodata`–specifies an initialized read-only data section.
- `bss`–specifies an uninitialized read-write data section.
- `sdata`–specifies a small initialized read-write data section.
- `sdata2`–specifies small initialized read-only data section.
- `sbss`–specifies a small uninitialized read-write data section.
- `debug`–specifies a debug section.
- `previous`–reverts to the previous section.
- `offset`–defines a record.
- `section`–specifies a section of any type.

text

```
.text
```

Specifies an executable code section. This must be in front of the actual code in a file.

data

```
.data
```

Specifies an initialized read-write data section.

rodata

```
.rodata
```

Specifies an initialized read-only data section.

bss

```
.bss
```

Specifies an uninitialized read-write data section.

sdata

```
.sdata
```

Specifies a small initialized read-write data section.

sdata2

```
.sdata2
```

Specifies a small initialized read-only data section.

sbss

```
.sbss
```

Specifies a small uninitialized read-write data section.

debug

```
.debug
```

Specifies a debug section. If you enable the debugger, the assembler automatically generates some debug information for your project. However, you use special directives in the debug section that provide the debugger with more detailed information. For more information on the debug directives, see “Debugging Directives.”

previous

```
.previous
```

Reverts to the previous section. This switch toggles between the current section and the previous section.

offset

```
.offset [expr]
```

Defines a record. The optional parameter *expr* specifies the initial location counter. The record definition extends until the start of the next section. Within a record, you can use only the following directives:

```
.equu               .set               .textequ
.align              .org               .space
.byte               .short             .long
.space              .ascii             .asciz
.float              .double
```

The data declaration directives (like `.byte` and `.short`) don’t allocate any storage. They just update the location counter.

Here is a sample record definition:

Listing 4.1 A record definition with the offset directive

```
          .offset
top:      .short  0
left:     .short  0
bottom:   .short  0
right:    .short  0
rectSize  .equ    *
```

section

```
.section name [,alignment],[type],[flags]
```

Specifies a section of name *name* with type *type*. Use this general form to create arbitrary relocatable sections, including sections to be loaded at an absolute address. These are the arguments to .section. Note that only the *name* argument is required.

- The *name* is the name of the section. It can be an symbol.
- The *type* and *flags* are both numeric, being the ELF section type/flags. The defaults for these fields are the type and flags

for the code section. The following example specifies a section named `vector` with an alignment of 4 bytes:

```
.section  vector,4
```

The possible ELF section types are defined in Table 4.1, and the possible ELF section flags are defined in Table 4.2.

Table 4.1 ELF Section Types

Type	Name
0	NULL
1	PROGBITS
2	SYMTAB
3	STRTAB
4	RELA
5	HASH
6	DYNAMIC
7	NOTE
8	NOBITS
9	REL
10	SHLIB
11	DYNSYM

Table 4.2 ELF Section Flags

Flag	Name
0x00000001	WRITE
0x00000002	ALLOC
0x00000004	EXECINSTR
0xF0000000	MASKPROC
0x10000000	GPREL

Scope Control Directives

The Metrowerks Assemblers provide directives that let you use equates outside the files they're defined in. Equates are symbols declared with `.set` or `.equ`, described in "Symbol Definition Directives". These are the directives:

- `global`–declares that equates are exported.
- `extern`–declares that equates are imported.
- `public`–declares that equates are public.

global

`.global` *equate* `[,`*equate*`]`...

Declares that the listed equates are exported, that is, available to other files. Equates are symbols declared with `.set` or `.equ`, described in "Symbol Definition Directives".

Use the `.extern` or `.public` directive to reference the symbols in another file.

You cannot export labels.

extern

`.extern` *equate* `[,`*equate*`]`...

Declares that the listed equates are imported: available to this file but defined in another file. Equates are symbols declared with `.set` or `.equ`, described in "Symbol Definition Directives".

Use the `.global` or `.public` directive to export the symbols from another file.

You cannot import labels.

public

`.public` *equate* `[,`*equate*`]`...

Declares that the listed equates are public. If the equates are already defined, the assembler exports them, that is, makes them available to other files. If the equates are *not* already defined, the assembler imports them, that is, makes them available to this file but defined in another file

Equates are symbols declared with `.set` or `.equ`, described in "Symbol Definition Directives". You cannot import labels.

Symbol Definition Directives

The following directives let you create equates:

- `set`–temporarily assigns a value to a symbol.
- equal sign (=)–temporarily assigns a value to a symbol and is available for compatibility with other assemblers.
- `equ`–permanently assigns a value to a symbol.
- `textequ`–defines a symbol that is substituted for some arbitrary text.

set

symbol `.set` *expr*

Temporarily assigns the value *expr* to the symbol *equate*. You may change *equate*'s value later. The symbol *equate* appears in the label field of the line, and the value *expr* appears in the operand field.

equal sign (=)

symbol `=` *expr*

Temporarily assigns the value *expr* to the symbol *symbol*. You may change *symbol*'s value later. The symbol *symbol* appears in the label field of the line, and the value *expr* appears in the operand field.

This directive is equivalent to `.set`, and is available only for compatibility with other company's assemblers. Some future assemblers may not support this directive.

equu

symbol `.equ` *expr*

Permanently assigns the value *expr* to the symbol *symbol*. You cannot change *symbol*'s value. The symbol *symbol* appears in the label field of the line, and the value *expr* appears in the operand field.

textequ

symbol `.textequ "`*string*`"`

Defines a symbol *symbol* that is substituted with any arbitrary text *string*. This directive helps you port existing code by letting you give new names to machine instructions, directives, and operands.

Whenever you use *symbol*, the assembler replaces it with *string* before performing any other processing on that source line. Here are some useful examples.

Listing 4.2 Some textequ examples

```
dc.b    .textequ    ".byte"
endc    .textequ    ".endif"
```

Data Declaration Directives

The Metrowerks Assembler has directives that initialize data. They are split into three sections:

- “Integer type declarations”
  - `byte`–declares an initialized block of bytes.
  - `short`–declares an initialized block of 16-bit short integers.
  - `long`–declares an initialized block of 32-bit short integers.
  - `space`–declares a block of zero-initialized bytes.
- “String type declarations”
  - `ascii`–declares a block of storage for a string.
  - `asciz`–declares a zero-terminated block of storage for a string.
- “Floating point type declarations”
  - `float`–declares an initialized block of 32-bit floating-point numbers.
  - `double`–declares an initialized block of 64-bit floating-point numbers.

Integer type declarations

These directives initialize blocks of integer data:

- `byte`–declares an initialized block of bytes.
- `short`–declares an initialized block of 16-bit short integers.
- `long`–declares an initialized block of 32-bit short integers.
- `space`–declares a block of zero-initialized bytes.
- `fill`–declares a block of zero-initialized bytes.

byte

[*label*] `.byte` *expr*[`,`*expr*]...

Declares an initialized block of bytes with the name *label*. The assembler allocates one 8-bit byte for each expression *expr*. Each expression must fit in the specified size.

short

[*label*] `.short` *expr*[`,`*expr*]...

Declares an initialized block of 16-bit short integers with the name *label*. The assembler allocates 16 bits for each expression *expr*. Each expression must fit in the specified size.

long

[*label*] `.long` *expr*[`,`*expr*]...

Declares an initialized block of 32-bit short integers with the name *label*. The assembler allocates 32 bits for each expression *expr*. Each expression must fit in the specified size.

space

[*label*] `.space` *expr*

Declares a block of zero-initialized bytes with the name *label*. The assembler allocates a block *expr* bytes long and initializes each byte to zero.

fill

[*label*] `.fill` *expr*

Declares a block of zero-initialized bytes with the name *label*. The assembler allocates a block *expr* bytes long and initializes each byte to zero.

String type declarations

These directives initialize blocks of character data:

- `ascii`–declares a block of storage for a string.
- `asciz`–declares a zero-terminated block of storage for a string.

Note that a string can also contain any of these escape sequences.

Table 4.3 Escape sequences

Sequence	Description
`\b`	Backspace
`\n`	Line feed (ASCII character 10)
`\r`	Return (ASCII character 13)
`\t`	Tab
`\"`	Double quote
`\\`	Backslash
`\nnn`	Octal value of \nnn

ascii

```
[label]  .ascii "string"
```

Declares a block of storage for the string *string* with the name *label*. The assembler allocates an 8-bit byte for each character in string.

asciz

```
[label]  .asciz "string"
```

Declares a zero-terminated block of storage for the string *string* with the name *label*. The assembler allocates an 8-bit byte for each character in string, and then allocates an extra block at the end that's initialized to zero.

Floating point type declarations

These directives initialize blocks of floating-point data:

- `float`–declares an initialized block of 32-bit floating-point numbers.
- `double`–declares an initialized block of 64-bit floating-point numbers.

float

[*label*] `.float` *value*[,*value*]…

Declares an initialized block of 32-bit floating-point numbers with the name *label*. The assembler allocates 32 bits for each value *value*. Each value must fit in the specified size.

double

[*label*] `.double` *value*[,*value*]…

Declares an initialized block of 64-bit floating-point numbers with the name *label*. The assembler allocates 64 bits for each value *value*. Each value must fit in the specified size.

Assembler Control Directives

These directives let you control how the assembler emits code:

- `align`–aligns the location counter to the next multiple of an expression.
- `endian`–specifies the byte ordering for the target processor.
- `error` –prints an error message.
- `include`–causes the assembler to switch input to another file.
- `pragma`–allows you to enable and disable certain code generation capabilities.
- `org`–changes the location counter.
- `option`–sets various assembler options.

align

```
.align expr
```

Aligns the location counter to the next multiple of the expression *expr*. The expression *expr* must be a power of 2, such as 2, 4, 8, 16, or 32.

endian

```
.endian big | little
```

Specifies the byte ordering for the target processor. You can use this directive only on processors that let you change the byte ordering.

error

```
.error "error"
```

Prints *error* to the Errors & Warnings window in the CodeWarrior IDE.

include

`.include` *filename*

Causes the assembler to switch input to *filename*. The assembler takes input from the specified file until the end of the file is reached. Then the assembler continues to take input from the assembly statement line that follows the `.include` directive.

The file specified by *filename* can have an `.include` directive for another file.

pragma

`.pragma` *pragma-type setting*

Tells the assembler to assemble the code using a given pragma setting. Refer to the *C Compiler Reference* for a list of relevant `pragma` statements.

org

`.org` *expr*

Changes the location counter to *expr*. The addresses of the following assembly statements start at the new value of the location counter. The value of *expr* must be greater than the current value of the location counter.

option

`.option` *keyword setting*

Sets the assembler options, as described in the table below. Specifying *reset* sets the option to it's previous setting. Using *reset* a second time resets the option to the setting before the current setting.

Table 4.4 Option keywords

This keyword	Does this
`alignment off|on|reset`	Controls whether data is aligned on natural boundary. This does not correspond any option in the settings panel.
`branchsize 8|16|32`	Specifies the size of forward branch displacement. This is allowed only for the x86 and 68K assemblers. This does not correspond any option in the settings panel
`colon off|on|reset`	Specifies whether labels must end with a colon (:). If it's `on`, every label needs a colon. If it's `off`, a labels doesn't need a colon if it starts in the first column. This corresponds to the **Labels must end with a ':'** option, described in "Labels must end with ':'."
`space off|on|reset`	Specifies whether space allowed in operand field. If it's `on`, operand fields may contain spaces. If it's `off`, a space in the operand field signals the start of a comment. This corresponds to the **Allow space in operand field** option, described in "Allow space in operand field."
`period off|on|reset`	Specifies whether a period (.) is required in directive names. If it's on, each directive must start with a period. If it's off, directives don't need to start with periods. This corresponds to the **Directives begin with '.'** option, described in "Directives begin with '.'."
`case off|on|reset`	Specifies where identifiers are case sensitive. If it's on, identifiers are case sensitive. If it's off, identifiers aren't case sensitive. This corresponds to the **Case sensitive identifiers** option, described in "Case sensitive identifiers."
`no_at_macros off | on`	If true, don't allow macros which use $AT. If false, warn if user uses $AT.

You can prevent the assembler from inserting a NOP (no operation) instruction after jumps and branches, and instead substitute the instruction of your choice. To do this, specify `.option reorder off` in the standalone assembler.

The standalone assembler inserts the NOP by default. However, in the inline assembler, NOP won’t be inserted for you.

Debugging Directives

These directives are allowed only in the `.debug` section of an assembly file. If you enable the debugger, the assembler automatically generates some debug information for your project. However, you can use these directives in the debug section to provide the debugger with more detailed information.

- `file`–writes debugging information to a specified output file.
- `function`–specifies information on a subroutine.
- `line`–specifies the absolute line number for the following code.
- `size`–specifies the length of a symbol.
- `type`–specifies whether a symbol is a function or object.

file

`.file "`*filename*`"`

Writes the debugging information for this file into *filename*. If this option isn't used, the debugging information is written to the project file.

function

`.function "`*func*`",` *label*`,` *length*

Specifies that the subroutine *func* begins at *label* and is *length* bytes long.

line

`.line` *number*

Specifies the absolute line number in the current source file which generated the following code or data. The first line in the file is numbered `1`.

size

```
.size   symbol, expr
```

Specifies that *symbol* is of *expr* bytes long.

type

```
.type   symbol, type
```

Specifies that *symbol* is of type *type*, where type can be either `@function` or `@object`.

5

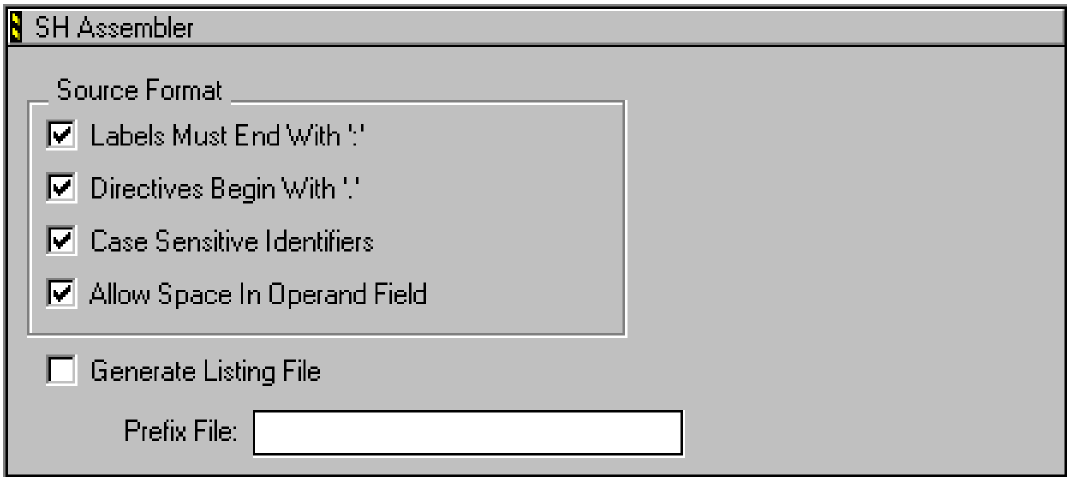
SH Assembler Settings

This chapter describes the options you can set for the Metrowerks Assemblers.

Assembler Settings Overview

There are several different assemblers available, one for each target processor family. Each assembler has several options you control through a settings panel. To modify the settings for an assembler, choose Project Settings on the Edit menu. In the resulting dialog box, select the name of the assembler to see its settings panel.

Figure 5.1 Assembler settings panel for SH Assembler



SH Assembler

Source Format

☑ Labels Must End With ':'

☑ Directives Begin With '.'

☑ Case Sensitive Identifiers

☑ Allow Space In Operand Field

□ Generate Listing File

Prefix File:

All of the settings panels are very similar to that shown in Figure 5.1, which shows the SH Assembler panel. The Source format section is the same for all.

The Assembler Settings Panel

The individual settings available to you are:

- Labels must end with ':'
- Directives begin with '.'
- Case sensitive identifiers
- Allow space in operand field
- Generate listing file
- Prefix file

Labels must end with ':'

The **Labels must end with ':'** option lets you choose whether labels must end in a colon (:). If this option is on, a label must ends in a colon (:) and can begin in any column. If this option is off, a symbol is a label if it starts in column 1 *or* if it ends with a colon (:). This option is especially useful if you're porting existing code that doesn't follow this convention. For more information on labels, "Symbol Syntax."

This option is on by default. It corresponds to the `colon` parameter of the `.option` directive, described in "option."

Directives begin with '.'

The **Directives begin with '.'** option lets you choose whether you must put a period at the beginning of each directive name. If this option is on, a directive must begin with a period (.). If you turn this option off, you can leave out the period. For more information on directives, see "Using Directives Overview."

This option is on by default. It corresponds to the `period` parameter of the `.option` directive, described in "option."

Case sensitive identifiers

The **Case sensitive identifiers** option lets you choose whether symbols are case-sensitive. If the option is on, symbols are case sensitive, so `SYM1`, `sym1`, and `Sym1` are three different symbols, for example. If the option is off, symbols are *not* case-sensitive, so `SYM1`, `sym1`, and `Sym1` are the same symbol, for example. For more information on symbols, see “Symbol Syntax.”

Note that instruction, directive, and macro names are always case insensitive, regardless of this option’s setting.

This option is on by default. It corresponds to the `case` parameter of the `.option` directive, described in “option.”

Allow space in operand field

The **Allow space in operand** option lets you choose if you can start a comment with a space in the operand field. If you turn this option on, spaces in the operand field are allowed. If you turn off this option, it ignores any text between a space character in the operand field and the end of the line. For more information on comments, see “Statement Syntax.”

This option is on by default. It corresponds to the `space` parameter of the `.option` directive, described in “option.”

Generate listing file

The **Generate listing file** option creates a text file that lets you compare your source code with the machine code the assembler produced. If you turn this option on, it creates a listing file using the source name and ‘`.list`’. For example, `test.asm` becomes `test.asm.list`. If you turn this option off, it doesn’t create a listing file.

This option is off by default.

Prefix file

The **Prefix file** field lets you specify a file that the assembler processes before every assembly file in your project. It’s as though you put the same .include directive at the beginning of every assembly file.

This field is blank by default.

Index

CodeWarrior

SH Assembler Reference

Credits

writing lead: Roger Wong

other writers: BitHead, John Roseborough, Jeff Mattson, L. Frank Turovich

engineering: Matt Cole

frontline warriors: Jim Trudeau, Eric Clapton

metrowerks®

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C and C++ compilers
Pascal Compilers Reference	Information on the Pascal and Object Pascal compilers
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
SH Assembler Guide	Stand-alone assembler manual for SH processors
Command-Line Reference	Command-line options for CodeWarrior compilers
Library Documentation	
MSL C Reference	Function reference for the Metrowerks standard C library
MSL C++ Reference	Function reference for the Metrowerks standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
The PowerPlant Book	Guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting Java	How to use CodeWarrior to program for the Java virtual machine
Targeting Mac	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting Palm	How to use CodeWarrior to program for Palm OS
Targeting PlayStation	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting Windows	How to use CodeWarrior to program for Windows 95/98/NT