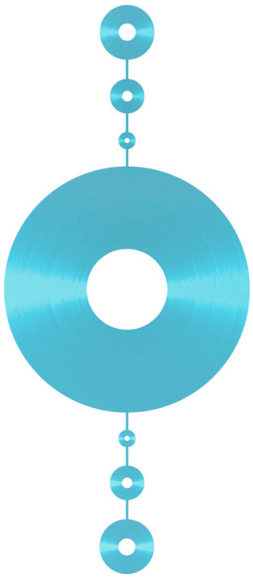
IAR Assembler™

Reference Guide

for the Renesas
SH Microcomputer Family



ASH-2

IAR SYSTEMS

EDITION NOTICE

Second edition: February 2010

Part number: ASH-2

This guide applies to version 2.x of the IAR Embedded Workbench® IDE for SH.

Internal reference: AFE2, M2, IJOA.

Contents

Tables

Preface

Welcome to the IAR Assembler for SH Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the IAR Assembler for SH to develop your application according to your requirements.

Who should read this guide

You should read this guide if you plan to develop an application, or part of an application, using assembler language for the SH microprocessor and need to get detailed reference information on how to use the IAR Assembler for SH. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the SH microprocessor. Refer to the documentation from Renesas for information about the SH microprocessor
- General assembler language programming
- Application development for embedded systems
- The operating system of your host computer.

How to use this guide

When you first begin using the IAR Assembler for SH, you should read the chapter *Introduction to the IAR Assembler for SH* in this reference guide.

If you are an intermediate or advanced user, you can focus more on the reference chapters that follow the introduction.

If you are new to using the IAR Systems toolkit, we recommend that you first read the initial chapters of the *IAR Embedded Workbench IDE User Guide*. They give product overviews, and tutorials that can help you get started.

What this guide contains

Below is a brief outline and summary of the chapters in this guide.

- *Introduction to the IAR Assembler for SH* provides programming information. It also describes the source code format, and the format of assembler listings.
- *Assembler options* first explains how to set the assembler options from the command line and how to use environment variables. It then gives an alphabetical summary of the assembler options, and contains detailed reference information about each option.
- *Assembler operators* gives a summary of the assembler operators, arranged in order of precedence, and provides detailed reference information about each operator.
- *Assembler directives* gives an alphabetical summary of the assembler directives, and provides detailed reference information about each of the directives, classified into groups according to their function.
- *Pragma directives* describes the pragma directives available in the assembler.
- *Diagnostics* contains information about the formats and severity levels of diagnostic messages.

Other documentation

The complete set of IAR Systems development tools for the SH microprocessor is described in a series of guides and online help files. For information about:

- Using the IAR Embedded Workbench® IDE with the IAR C-SPY® Debugger, refer to the *IAR Embedded Workbench IDE User Guide*
- Programming for the IAR C/C++ Compiler for SH and using the IAR ILINK Linker, refer to the *IAR C/C++ Development Guide for SH*
- Using the IAR DLIB Library, refer to the online help system

All of these guides are delivered in hypertext PDF or HTML format on the installation media.

Document conventions

When, in this text, we refer to the programming language C, the text also applies to C++, unless otherwise stated.

When referring to a directory in your product installation, for example `sh\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.`*`n`*`\sh\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file.
`[option]`	An optional part of a command.
`a|b|c`	Alternatives in a command.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems® referred to in this guide:

Brand name	Generic term
IAR Embedded Workbench® for SH	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for SH	the IDE
IAR C-SPY® Debugger for SH	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for SH	the compiler

Table 2: Naming conventions used in this guide

Brand name	Generic term
IAR Assembler™ for SH	the assembler
IAR ILINK™ Linker	ILINK, the linker
IAR DLIB Library™	the DLIB library

Table 2: Naming conventions used in this guide (Continued)

Introduction to the IAR Assembler for SH

This chapter contains these sections:

- Introduction to assembler programming
- Modular programming
- External interface details
- Source format
- Assembler instructions
- Expressions, operands, and operators
- List file format
- Programming hints.

Introduction to assembler programming

Even if you do not intend to write a complete application in assembler language, there might be situations where you find it necessary to write parts of the code in assembler, for example, when using mechanisms in the SH microprocessor that require precise timing and special instruction sequences.

To write efficient assembler applications, you should be familiar with the architecture and instruction set of the SH microprocessor. Refer to the hardware documentation from Renesas for syntax descriptions of the instruction mnemonics.

GETTING STARTED

To ease the start of the development of your assembler application, you can:

- Work through the tutorials—especially the one about mixing C and assembler modules—that you find in the *IAR Embedded Workbench IDE User Guide*
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the *IAR C/C++ Development Guide for SH*

- In the IAR Embedded Workbench IDE, you can base a new project on a *template* for an assembler project.

Modular programming

It is widely accepted that modular programming is a prominent feature of good software design. If you structure your code in small modules—in contrast to one single monolith—you can organize your application code in a logical structure, which makes the code easier to understand, and which aids:

- efficient program development
- reuse of modules
- maintenance.

The IAR development tools provide different facilities for achieving a modular structure in your software.

Typically, you write your assembler code in assembler source files; each file becomes a named *module*. If you divide your source code into many small source files, you will get many small modules. You can divide each module further into different subroutines.

A *section* is a logical entity containing a piece of data or code that should be mapped to a physical location in memory. Use the section control directives to place your code and data in sections. A section is *relocatable*. An address for a relocatable section is resolved at link time. Sections let you control how your code and data is placed in memory. A section is the smallest linkable unit, which allows the linker to include only those units that are referred to.

If you are working on a large project you will soon accumulate a collection of useful routines that are used by several of your applications. To avoid ending up with a huge amount of small object files, collect modules that contain such routines in a *library* object file. Note that a module in a library is always conditionally linked. In the IDE, you can set up a library project, to collect many object files in one library. For an example, see the tutorials in the *IAR Embedded Workbench IDE User Guide*.

To summarize, your software design benefits from modular programming, and to achieve a modular structure you can:

- Create many small modules, one per source file
- In each module, divide your assembler source code into small subroutines (corresponding to *functions* on the C level)
- Divide your assembler source code into *sections*, to gain more precise control of how your code and data finally is placed in memory

- Collect your routines in libraries, which means that you can reduce the number of object files and make the modules conditionally linked.

External interface details

This section provides information about how the assembler interacts with its environment.

You can use the assembler either from the IDE or from the command line. Refer to the *IAR Embedded Workbench IDE User Guide* for information about using the assembler from the IDE.

ASSEMBLER INVOCATION SYNTAX

The invocation syntax for the assembler is:

```
iasmsh [options][sourcefile][options]
```

For example, when assembling the source file `prog.s`, use this command to generate an object file with debug information:

```
iasmsh prog --debug
```

By default, the assembler recognizes the filename extensions `s`, `asm`, and `msa` for source files. The default filename extension for assembler output is `o`.

Generally, the order of options on the command line, both relative to each other and to the source filename, is *not* significant. However, there is one exception: when you use the `-I` option, the directories are searched in the same order that they are specified on the command line.

If you run the assembler from the command line without any arguments, the assembler version number and all available options including brief descriptions are directed to `stdout` and displayed on the screen.

PASSING OPTIONS

You can pass options to the assembler in three different ways:

- Directly from the command line

  Specify the options on the command line after the `iasmsh` command; see *Assembler invocation syntax*, page 3.
- Via environment variables

  The assembler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 4.
- Via a text file by using the `-f` option; see *-f*, page 27.

For general guidelines for the option syntax, an options summary, and a detailed description of each option, see the *Assembler options* chapter.

ENVIRONMENT VARIABLES

Assembler options can also be specified in the `IASMSH` environment variable. The assembler automatically appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every assembly.

You can use these environment variables with the IAR Assembler for SH:

Environment variable	Description
`IASMSH`	Specifies command line options; for example: `set IASMSH=-la . --warnings_are_errors`
`IASMSH_INC`	Specifies directories to search for include files; for example: `set IASMSH_INC=c:\myinc\`

Table 3: Assembler environment variables

For example, setting this environment variable always generates a list file with the name `temp.lst`:

```
set IASMSH=-l temp.lst
```

For information about the environment variables used by the compiler and linker, see the *IAR C/C++ Development Guide for SH*.

ERROR RETURN CODES

When using the assembler from within a batch file, you might have to determine whether the assembly was successful to decide what step to take next. For this reason, the assembler returns these error return codes:

Return code	Description
`0`	Assembly successful, warnings might appear.
`1`	Warnings occurred, provided that the option `--warnings_affect_exit_code` was used.
`2`	Non-fatal errors or fatal assembly errors occurred (making the assembler abort).
`3`	Crashing errors occurred.

Table 4: Assembler error return codes

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

label	A definition of a label, which is a symbol that represents an address. If the label starts in the first column—that is, at the far left on the line—the : (colon) is optional.
operation	An assembler instruction or directive. This must not start in the first column—there must be some whitespace to the left of it.
operands	An assembler instruction or directive can have zero, one, or more operands. The operands are separated by commas. An operand can be: • a constant representing a numeric value or an address • a symbolic name representing a numeric value or an address (where the latter also is referred to as a label) • a floating-point constant • a register • a predefined symbol • the program location counter (PLC) • an expression.
comment	Comment, preceded by a ; (semicolon) C or C++ comments are also allowed.

The components are separated by spaces or tabs.

A source line can not exceed 2047 characters.

Tab characters, ASCII 09H, are expanded according to the most common practice; i.e. to columns 8, 16, 24 etc. This affects the source code output in list files and debug information. Because tabs might be set up differently in different editors, do not use tabs in your source files.

Assembler instructions

The IAR Assembler for SH supports the syntax for assembler instructions as described in the chip manufacturer's hardware documentation. It complies with the requirement of the SH architecture on word alignment. Any instructions in a code section placed on an odd address results in an error.

Expressions, operands, and operators

Expressions consist of expression operands and operators.

The assembler accepts a wide range of expressions, including both arithmetic and logical operations. All operators use 32-bit two's complement integers. Range checking is performed if a value is used for generating code.

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see also *Assembler operators*, page 35.

These operands are valid in an expression:

- Constants for data or addresses, excluding floating-point constants.
- Symbols—symbolic names—which can represent either data or addresses, where the latter also is referred to as *labels*.
- The program location counter (PLC), `$` (dollar).

The operands are described in greater detail on the following pages.

Note: You cannot have two symbols in one expression, or any other complex expression, unless the expression can be resolved at assembly time. In this case, the assembler generates an error.

INTEGER CONSTANTS

Because all IAR Systems assemblers use 32-bit two's complement internal arithmetic, integers have a (signed) range from -2147483648 to 2147483647.

Constants are written as a sequence of digits with an optional – (minus) sign in front to indicate a negative number.

Commas and decimal points are not permitted.

The following types of number representation are supported:

Integer type	Example
Binary	`1010b`
Octal	`1234q`
Decimal	`1234`, `-1`
Hexadecimal	`0FFFFh`, `0xFFFF`

Table 5: Integer constant formats

Note: Both the prefix and the suffix can be written with either uppercase or lowercase letters.

ASCII CHARACTER CONSTANTS

ASCII constants can consist of any number of characters enclosed in single or double quotes. Only printable characters and spaces can be used in ASCII strings. If the quote character itself will be accessed, two consecutive quotes must be used:

Format	Value
`'ABCD'`	`ABCD` (four characters).
`"ABCD"`	`ABCD'\0'` (five characters the last ASCII null).
`'A''B'`	`A'B`
`'A'''`	`A'`
`''''` (4 quotes)	`'`
`''` (2 quotes)	Empty string (no value).
`""` (2 double quotes)	Empty string (an ASCII null character).
`\'`	`'`, for quote within a string, as in `'I\'d love to'`
`\\`	`\`, for `\` within a string
`\"`	`"`, for double quote within a string

Table 6: ASCII character constant formats

FLOATING-POINT CONSTANTS

The IAR Assembler for SH accepts floating-point values as constants and convert them into IEEE single-precision (signed 32-bit) floating-point format and double-precision (signed 64-bit), or fractional format.

Floating-point numbers can be written in the format:

`[+|-][`*`digits`*`].[`*`digits`*`][{E|e}[+|-]`*`digits`*`]`

This table shows some valid examples:

Format	Value
10.23	1.023×10^{1}
1.23456E-24	1.23456×10^{-24}
1.0E3	1.0×10^{3}

Table 7: Floating-point constants

Spaces and tabs are not allowed in floating-point constants.

Note: Floating-point constants do not give meaningful results when used in expressions.

When a fractional format is used—for example, `DQ15`—the range that can be represented is `-1.0 <= x < 1.0`. Any value outside that range is silently saturated into the maximum or minimum value that can be represented.

If the word length of the fractional data is `n`, the fractional number will be represented as the 2-complement number: `x * 2^(n-1)`.

TRUE AND FALSE

In expressions a zero value is considered FALSE, and a non-zero value is considered TRUE.

Conditional expressions return the value 0 for FALSE and 1 for TRUE.

SYMBOLS

User-defined symbols can be up to 255 characters long, and all characters are significant. Depending on what kind of operation a symbol is followed by, the symbol is either a data symbol or an address symbol where the latter is referred to as a label. A symbol before an instruction is a label and a symbol before, for example the `EQU` directive, is a data symbol. A symbol can be:

- absolute—its value is known by the assembler
- relocatable—its value is resolved at link time.

Symbols must begin with a letter, a–z or A–Z, ? (question mark), or _ (underscore). Symbols can include the digits 0–9 and `$` (dollar).

Case is insignificant for built-in symbols like instructions, registers, operators, and directives. For user-defined symbols, case is by default significant but can be turned on and off using the **Case sensitive user symbols** (`--case_insensitive`) assembler option. See *--case_insensitive*, page 20 for additional information.

Use the symbol control directives to control how symbols are shared between modules. For example, use the `PUBLIC` directive to make one or more symbols available to other modules. The `EXTERN` directive is used for importing an untyped external symbol.

Note that symbols and labels are byte addresses. For additional information, see *Generating a lookup table*, page 88.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

The assembler keeps track of the start address of the current instruction. This is called the *program location counter.*

If you must refer to the program location counter in your assembler source code, use the $ (dollar) sign. For example:

```
        bra     $       ; Loop forever
```

REGISTER SYMBOLS

This table shows the existing predefined register symbols:

Name	Size	Description
R0–R14	32 bits	General purpose registers
SP (R15)	32 bits	Stack pointer
FR0–FR15	32 bits	General purpose floating-point registers. Only for SH-2A microprocessors with a hardware FPU.
DR0, DR2, DR4 ... DR14	64 bits	Double-precision floating-point registers. Only for SH-2A microprocessors with a hardware FPU.
PC	32 bits	Program counter
PR	32 bits	Program return register
SR	32 bits	Status register
GBR	32 bits	Global base register
VBR	32 bits	Vector base register
TBR	32 bits	Jump table base register
MACH	32 bits	Multiply and accumulate register high
MACL	32 bits	Multiply and accumulate register low
FPUL	32 bits	Floating-point communication register. Only for SH-2A microprocessors with a hardware FPU.
FPSCR	32 bits	Floating-point status register. Only for SH-2A microprocessors with a hardware FPU.

Table 8: Predefined register symbols

Note: The IAR Assembler for SH does not accept *register pairs*.

PREDEFINED SYMBOLS

The IAR Assembler for SH defines a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in preprocessor directives or include them in the assembled code. The strings returned by the assembler are enclosed in double quotes.

These predefined symbols are available:

Symbol	Value
`__IASMSH__`	An integer that is set to `1` when the code is assembled with the IAR Assembler for SH.
`__BUILD_NUMBER__`	A unique integer that identifies the build number of the assembler currently in use. The build number does not necessarily increase with an assembler that is released later.
`__CODE_MODEL__`	An integer that identifies the code model in use. The symbol reflects the `--code_model` option and is defined to `__CODE_MODEL_SMALL__`, `__CODE__MODEL_MEDIUM__`, `__CODE_MODEL_LARGE__`, or `__CODE_MODEL_HUGE__`. These symbolic names can be used when testing the `__CODE_MODEL__` symbol.
`__CORE__`	An integer that identifies the chip core in use. The symbol reflects the `--core` option and is defined to `__SH2A__` or `__SH2AFPU__`. These symbolic names can be used when testing the `__CORE__` symbol.
`__DATA_MODEL__`	An integer that identifies the data model in use. The symbol reflects the `--data_model` option and is defined to `__DATA_MODEL_SMALL__`, `__DATA_MODEL_MEDIUM__`, `__DATA_MODEL_LARGE__`, or `__DATA_MODEL_HUGE__`. These symbolic names can be used when testing the `__DATA_MODEL__` symbol.
`__DATE__`	The current date in dd/Mmm/yyyy format (string).
`__DEFAULT_CODE_SECTION__`	A symbol that identifies the default memory section for code. The symbol reflects the `--code_model` option and is defined to `.code16.txt`, `.code20.txt`, `.code28.txt`, or `.code32.txt`. These symbolic names can be used when testing the `__DEFAULT_CODE_SECTION__` symbol.
`__DOUBLE__`	An integer that identifies the size of the data type `double`. The symbol reflects the `--double` option and is defined to `32` or `64`.
`__FILE__`	The name of the current source file (string).

Table 9: Predefined symbols

Symbol	Value
__IAR_SYSTEMS_ASM__	IAR assembler identifier (number). The current value is 7. Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was assembled by an assembler from IAR Systems.
__LINE__	The current source line number (number).
__SUBVERSION__	An integer that identifies the subversion number of the assembler version number, for example 3 in 1.2.3.4.
__TIME__	The current time in `hh:mm:ss` format (string).
__VER__	The version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 9: Predefined symbols (Continued)

Including symbol values in code

Several data definition directives make it possible to include a symbol value in the code. These directives define values or reserve memory. To include a symbol value in the code, use the symbol in the appropriate data definition directive.

For example, to include the time of assembly as a string for the program to display:

```
            name    timeOfAssembly
            extern  printStr
            public  printTime
            section __DEFAULT_CODE_SECTION__:CODE(2)

            CODE
printTime:  mova    time,R0         ; Load address of time
                                    ; string in R0.
            mov.l   ct_printStr,R1  ; Move printStr function
                                    ; address to R1
            jsr     @R1             ; Call string output routine.
            nop
            rts/n
            section __DEFAULT_CODE_SECTION__:CODE(2)
            DATA
ct_printStr dc32    printStr
time        dc8     __TIME__        ; String representing the
                                    ; time of assembly.
            end
```

Testing symbols for conditional assembly

To test a symbol at assembly time, use one of the conditional assembly directives. These directives let you control the assembly process at assembly time.

For example, if you want to assemble separate code sections depending on whether you are using an old assembler version or a new assembler version, do as follows:

```
#if (__VER__ > 300)                ; New assembler version
;…
;…
#else                              ; Old assembler version
;…
;…
#endif
```

See *Conditional assembly directives*, page 66.

ABSOLUTE AND RELOCATABLE EXPRESSIONS

Depending on what operands an expression consists of, the expression is either *absolute* or *relocatable*. Absolute expressions are those expressions that only contain absolute symbols or relocatable symbols that cancel each other out.

Expressions that include symbols in relocatable sections cannot be resolved at assembly time, because they depend on the location of sections. These are referred to as relocatable expressions.

Such expressions are evaluated and resolved at link time, by the IAR ILINK Linker. They can only be built up out of a maximum of one symbol reference and an offset after the assembler has reduced it.

For example, a program could define the section `DATA` as follows:

```
            name    simpleExpressions
            extern  size
            section DATA:DATA
            DATA
first       dc8     5                ; An absolute expression.

            dc8     first            ; Examples of some legal
            dc8     first + 1        ; relocatable expressions.
            dc8     size + 1
            end
```

Note: At assembly time, there is no range check. The range check occurs at link time and, if the values are too large, there is a linker error.

EXPRESSION RESTRICTIONS

Expressions can be categorized according to restrictions that apply to some of the assembler directives. One such example is the expression used in conditional statements like `IF`, where the expression must be evaluated at assembly time and therefore cannot contain any external symbols.

The following expression restrictions are referred to in the description of each directive they apply to.

No forward

All symbols referred to in the expression must be known, no forward references are allowed.

No external

No external references in the expression are allowed.

Absolute

The expression must evaluate to an absolute value; a relocatable value (section offset) is not allowed.

Fixed

The expression must be fixed, which means that it must not depend on variable-sized instructions. A variable-sized instruction is an instruction that might vary in size depending on the numeric value of its operand.

List file format

The format of an assembler list file is as follows:

HEADER

The header section contains product version information, the date and time when the file was created, and which options were used.

BODY

The body of the listing contains the following fields of information:

- The line number in the source file. Lines generated by macros, if listed, have a . (period) in the source line number field.

- The address field shows the location in memory, which is relative. The notation is hexadecimal.
- The data field shows the data generated by the source line. The notation is hexadecimal. Unresolved values are represented by ..... (periods), where two periods signify one byte. These unresolved values are resolved during the linking process.
- The assembler source line.

SUMMARY

The *end* of the file contains a summary of errors and warnings that were generated.

SYMBOL AND CROSS-REFERENCE TABLE

When you specify the **Include cross-reference** option, or if the `LSTXRF+` directive was included in the source file, a symbol and cross-reference table is produced.

This information is provided for each symbol in the table:

Information	Description
Symbol	The symbol's user-defined name.
Mode	ABS (Absolute), or REL (Relocatable).
Sections	The name of the section that this symbol is defined relative to.
Value/Offset	The value (address) of the symbol within the current module, relative to the beginning of the current section.

Table 10: Symbol and cross-reference table

Programming hints

This section gives hints on how to write efficient code for the assembler. For information about projects including both assembler and C or C++ source files, see the *IAR C/C++ Development Guide for SH*.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several SH devices are included in the IAR Systems product package, in the `\sh\inc` directory. These header files define the processor-specific special function registers (SFRs) and interrupt vector numbers.

The header files are intended to be used also with the IAR C/C++ Compiler for SH, and they are suitable to use as templates when creating new header files for other SH devices.

If any assembler-specific additions are needed in the header file, you can easily add these in the assembler-specific part of the file:

```
#ifdef __IAR_SYSTEMS_ASM__
  ; Add your assembler-specific defines here.
#endif
```

USING C-STYLE PREPROCESSOR DIRECTIVES

The C-style preprocessor directives are processed before other assembler directives. Therefore, do not use preprocessor directives in macros and do not mix them with assembler-style comments. For more information about comments, see *Assembler control directives*, page 89.

Assembler options

This chapter first explains how to set the options from the command line, and gives an alphabetical summary of the assembler options. It then provides detailed reference information for each assembler option.

The *IAR Embedded Workbench IDE User Guide* describes how to set assembler options in the IAR Embedded Workbench® IDE, and gives reference information about the available options.

Setting command line assembler options

To set assembler options from the command line, include them on the command line after the `iasmsh` command, either before or after the source filename. For example, when assembling the source file `prog.s`, use this command to generate an object file with debug information:

```
iasmsh prog --debug
```

Some options accept a filename, included after the option letter with a separating space. For example, to generate a listing to the file `prog.lst`:

```
iasmsh prog -l prog.lst
```

Some other options accept a string that is not a filename. The string is included after the option letter, but without a space. For example, to define a symbol:

```
iasmsh prog -DDEBUG=1
```

Generally, the order of options on the command line, both relative to each other and to the source filename, is *not* significant. However, there is one exception: when you use the `-I` option, the directories are searched in the same order as they are specified on the command line.

Notice that a command line option has a *short* name and/or a *long* name:

- A short option name consists of one character, with or without parameters. You specify it with a single dash, for example `-r`.
- A long name consists of one or several words joined by underscores, and it can have parameters. You specify it with double dashes, for example `--debug`.

SPECIFYING PARAMETERS

When a parameter is needed for an option with a short name, you can specify it either immediately following the option or as the next command line argument.

For instance, you can specify an include file path of `\usr\include` either as:

```
-I\usr\include
```

or as

```
-I \usr\include
```

Note: You can use `/` instead of `\` as directory delimiter. A trailing backslash can be added to the last directory name, but is not required.

Additionally, output file options can take a parameter that is a directory name. The output file then receives a default name and extension.

When a parameter is needed for an option with a long name, you can specify it either immediately after the equal sign (`=`) or as the next command line argument, for example:

```
--diag_suppress=Pe0001
```

or

```
--diag_suppress Pe0001
```

Options that accept multiple values can be repeated, and can also have comma-separated values (without space), for example:

```
--diag_warning=Be0001,Be0002
```

The current directory is specified with a period (`.`), for example:

```
iasmsh prog -l .
```

A file specified by `-` (a single dash) is standard input or output, whichever is appropriate.

Note: When an option takes a parameter, the parameter cannot start with a dash (`-`) followed by another character. Instead you can prefix the parameter with two dashes (`--`). This example generates a list on standard output:

```
iasmsh prog -l ---
```

Summary of assembler options

This table summarizes the assembler options available from the command line:

Command line option	Description
`--case_insensitive`	Case-insensitive user symbols

Table 11: Assembler options summary

Command line option	Description
--code_model	Specifies the code model
--core	Specifies a CPU core
-D	Defines preprocessor symbols
--data_model	Specifies the data model
--debug	Generates debug information
--dependencies	Lists file dependencies
--diag_error	Treats these diagnostics as errors
--diag_remark	Treats these diagnostics as remarks
--diag_suppress	Suppresses these diagnostics
--diag_warning	Treats these diagnostics as warnings
--diagnostics_tables	Lists all diagnostic messages
--dir_first	Allows directives in the first column
--double	Specifies 32-bit or 64-bit doubles
--enable_multibytes	Enables support for multibyte characters
--error_limit	Specifies the allowed number of errors before the assembler stops
-f	Extends the command line
--header_context	Lists all referred source files
-I	Add search path for header file
-l	Generates list file
-M	Macro quote characters
--mnem_first	Allows mnemonics in the first column
--no_fragments	Disables section fragment handling
--no_path_in_file_macros	Removes the path from the return value of the symbols __FILE__ and __BASE_FILE__
--no_system_include	Disables the automatic search for system include files
--no_warnings	Disables all warnings
--no_wrap_diagnostics	Disables wrapping of diagnostic messages
-o	Sets object filename. Alias for --output.
--only_stdout	Uses standard output only
--output	Sets object filename
--predef_macros	Lists the predefined symbols.

Table 11: Assembler options summary (Continued)

Command line option	Description
`--preinclude`	Includes an include file before reading the source file
`--preprocess`	Preprocessor output to file
`-r`	Generates debug information. Alias for `--debug`.
`--remarks`	Enables remarks
`--silent`	Sets silent operation
`--system_include_dir`	Specifies the path for system include files
`--use_unix_directory_separators`	Uses / as directory separator in paths
`--warnings_affect_exit_code`	Warnings affect exit code
`--warnings_are_errors`	Treats all warnings as errors

Table 11: Assembler options summary (Continued)

Description of assembler options

The following sections give detailed reference information about each assembler option.

Note that if you use the page **Extra Options** to specify specific command line options, there is no check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--case_insensitive

`--case_insensitive`

Use this option to make user symbols case insensitive.

By default, case sensitivity is on. This means that, for example, `LABEL` and `label` refer to different symbols. Use `--case_insensitive` to turn case sensitivity off, in which case `LABEL` and `label` refer to the same symbol.

You can also use the assembler directives `CASEON` and `CASEOFF` to control case sensitivity for user-defined symbols. See *Assembler control directives*, page 89, for more information.

Note: The `--case_insensitive` option does not affect preprocessor symbols. Preprocessor symbols are always case sensitive, regardless of whether they are defined in the IDE or on the command line. See *Defining and undefining preprocessor symbols*, page 83.

Project>Options>Assembler >Language>User symbols are case sensitive

--code_model `--code_model{small|medium|large|huge}`

`small`	The Small code model is used
`medium`	The Medium code model is used
`large`	The Large code model is used
`huge` (default)	The Huge code model is used

Use this option to inform the assembler of which code model that is used. All modules of your application must use the same code model.

This option also defines the symbols `__CODE_MODEL__` and `__DEFAULT_CODE_SECTION__`. See *Predefined symbols*, page 9.

Project>Options>General Options>Target>Code model

--core `--core={sh2a|sh2afpu}`

`sh2a` (default)	SH-2A microprocessors without a hardware FPU
`sh2afpu`	SH-2A microprocessors with a hardware FPU

Use this option to specify the processor core for which the code will be generated. If you do not use the option to specify a core, the assembler assumes you are assembling for the SH-2A core without an FPU. Note that all modules of your application must use the same core.

The assembler supports all devices based on the SH-2A microprocessor core.

To set related options, select:

Project>Options>General Options>Target>Device

-D `-Dsymbol[=value]`

Defines a symbol to be used by the preprocessor with the name *`symbol`* and the value *`value`*. If no value is specified, 1 is used.

The `-D` option allows you to specify a value or choice on the command line instead of in the source file.

Example

You might want to arrange your source to produce either the test or production version of your program dependent on whether the symbol `TESTVER` was defined. To do this use include sections such as:

```
#ifdef  TESTVER
...     ; additional code lines for test version only
#endif
```

Then select the version required on the command line as follows:

Production version:	`iasmsh prog`
Test version:	`iasmsh prog -DTESTVER`

Alternatively, your source might use a variable that you must change often. You can then leave the variable undefined in the source, and use `-D` to specify the value on the command line; for example:

```
iasmsh prog -DFRAMERATE=3
```

Project>Options>Assembler>Preprocessor>Defined symbols

`--data_model` `--data_model{small|medium|large|huge}`

`small`	The Small data model is used
`medium`	The Medium data model is used
`large`	The Large data model is used
`huge` (default)	The Huge data model is used

Use this option to inform the assembler of which data model that is used. All modules of your application must use the same data model.

This option also defines the symbol `__DATA_MODEL__`. See *Predefined symbols*, page 9.

Project>Options>General Options>Target>Data model

`--debug, -r` `--debug`

`-r`

The `--debug` option makes the assembler generate debug information that allows a symbolic debugger such as the IAR C-SPY® Debugger to be used on the program.

to reduce the size and link time of the object file, the assembler does not generate debug information by default.

Project>Options>Assembler >Output>Generate debug information

`--dependencies` `--dependencies=[i][m] {`*`filename`*`|`*`directory`*`}`

When you use this option, each source file opened by the assembler is listed in a file. These modifiers are available:

Option modifier	Description
`i`	Include only the names of files (default)
`m`	Makefile style

Table 12: Generating a list of dependencies (--dependencies)

If a *`filename`* is specified, the assembler stores the output in that file.

If a *`directory`* is specified, the assembler stores the output in that directory, in a file with the extension `i`. The filename is the same as the name of the assembled source file, unless a different name was specified with the `-o` option, in which case that name is used.

To specify the working directory, replace *`directory`* with a period (`.`).

If `--dependencies` or `--dependencies=i` is used, the name of each opened source file, including the full path if available, is output on a separate line. For example:

```
c:\iar\product\include\stdio.h
d:\myproject\include\foo.h
```

If `--dependencies=m` is used, the output uses makefile style. For each source file, one line containing a makefile dependency rule is output. Each line consists of the name of the object file, a colon, a space, and the name of a source file. For example:

```
foo.o: c:\iar\product\include\stdio.h
foo.o: d:\myproject\include\foo.h
```

Example 1

To generate a listing of file dependencies to the file `listing.i`, use:

```
iasmsh prog --dependencies=i listing
```

Example 2

To generate a listing of file dependencies to a file called `listing.i` in the `mypath` directory, you would use:

```
iasmsh prog --dependencies \mypath\listing
```

Note: You can use both `\` and `/` as directory delimiters.

Example 3

An example of using `--dependencies` with gmake:

1. Set up the rule for assembling files to be something like:

```
%.o : %.c
      $(ASM) $(ASMFLAGS) $< --dependencies=m $*.d
```

   That is, besides producing an object file, the command also produces a dependent file in makefile style (in this example using the extension `.d`).

2. Include all the dependent files in the makefile using, for example:

```
-include $(sources:.c=.d)
```

   Because of the `-`, it works the first time, when the `.d` files do not yet exist.

This option is not available in the IDE.

`--diag_error` `--diag_error=`*tag*`,`*tag*`,...`

Use this option to classify diagnostic messages as errors.

An error indicates a violation of the assembler language rules, of such severity that object code is not generated, and the exit code will not be 0.

This example classifies warning `As001` as an error:

```
--diag_error=As001
```

Project>Options>Assembler >Diagnostics>Treat these as errors

`--diag_remark` `--diag_remark=`*tag*`,`*tag*`,...`

Use this option to classify diagnostic messages as remarks.

A remark is the least severe type of diagnostic message and indicates a source code construct that might cause strange behavior in the generated code.

This example classifies the warning `As001` as a remark:

```
--diag_remark=As001
```

Project>Options>Assembler >Diagnostics>Treat these as remarks

--diag_suppress

```
--diag_suppress=tag,tag,...
```

Use this option to suppress diagnostic messages. This example suppresses the warnings `As001` and `As002`:

```
--diag_suppress=As001,As002
```

Project>Options>Assembler >Diagnostics>Suppress these diagnostics

--diag_warning

```
--diag_warning=tag,tag,...
```

Use this option to classify diagnostic messages as warnings.

A warning indicates an error or omission that is of concern, but which does not cause the assembler to stop before the assembly is completed.

This example classifies the remark `As028` as a warning:

```
--diag_warning=As028
```

Project>Options>Assembler >Diagnostics>Treat these as warnings

--diagnostics_tables

```
--diagnostics_tables {filename|directory}
```

Use this option to list all possible diagnostic messages in a named file. This can be very convenient, for example, if you used a `#pragma` directive to suppress or change the severity level of any diagnostic messages, but forgot to document why.

This option cannot be given together with other options.

If a *filename* is specified, the assembler stores the output in that file.

If a *directory* is specified, the assembler stores the output in that directory, in a file with the name `diagnostics_tables.txt`. To specify the working directory, replace *directory* with a period (`.`).

Example 1

To output a list of all possible diagnostic messages to the file `diag.txt`, use:

```
--diagnostics_tables diag
```

Example 2

If you want to generate a table to a file `diagnostics_tables.txt` in the working directory, you could use:

```
--diagnostics_tables .
```

You can use both `\` and `/` as directory delimiters.

This option is not available in the IDE.

`--dir_first` `--dir_first`

The default behavior of the assembler is to treat all identifiers starting in the first column as labels.

Use this option to make directive names (without a trailing colon) that start in the first column to be recognized as directives.

Project>Options>Assembler >Language>Allow directives in first column

---

`--double` `--double={32|64}`

`32` (default)	32-bit doubles are used
`64`	64-bit doubles are used

Use this option to specify the precision used for representing the floating-point types `double` and `long double`. The precision can be either 32-bit or 64-bit. By default, the assembler assumes you are using 32-bit doubles.

Project>Options>General Options>Target>Size of type 'double'

`--enable_multibytes` `--enable_multibytes`

By default, multibyte characters cannot be used in assembler source code. If you use this option, multibyte characters in the source code are interpreted according to the host computer's default setting for multibyte support.

Multibyte characters are allowed in comments, in string literals, and in character constants. They are transferred untouched to the generated code.

Project>Options>Assembler>Language>Enable multibyte support

`--error_limit` `--error_limit=`*`n`*

Use the `--error_limit` option to specify the number of errors allowed before the assembler stops. By default, 100 errors are allowed. *`n`* must be a positive number; `0` indicates no limit.

Project>Options>Assembler >Diagnostics>Max number of errors

-f `-f filename`

Extends the command line with text read from the specified file. Notice that there must be a space between the option itself and the filename.

The `-f` option is particularly useful if there are many options which are more conveniently placed in a file than on the command line itself.

Example

To run the assembler with further options taken from the file `extend.xcl`, use:

```
iasmsh prog -f extend.xcl
```

To set this option, use:

Project>Options>Assembler>Extra Options

--header_context `--header_context`

Occasionally, you must know which header file that was included from what source line, to find the cause of a problem. Use this option to list, for each diagnostic message, not only the source position of the problem, but also the entire include stack at that point.

This option is not available in the IDE.

-I `-I*path*`

Use this option to specify paths to be used by the preprocessor, by adding the `#include` file search prefix *`path`*.

By default, the assembler searches for `#include` files only in the current working directory and in the paths specified in the `ASH_INC` environment variable. The `-I` option allows you to give the assembler the names of directories which it will also search if it fails to find the file in the current working directory.

Example

For example, using the options:

```
-Ic:\global\ -Ic:\thisproj\headers\
```

and then writing:

```
#include "asmlib.hdr"
```

in the source, makes the assembler search first in the current directory, then in the directory `c:\global\`, and then in the directory `C:\thisproj\headers\`. Finally,

the assembler searches the directories specified in the ASH_INC environment variable, provided that this variable is set.

Project>Options>Assembler >Preprocessor>Additional include directories

---

`-l` `-l[a][d][e][m][o][x][N] {`*`filename`*`|`*`directory`*`}`

By default, the assembler does not generate a listing. Use this option to generate a listing to a file.

You can choose to include one or more of the following types of information:

Command line option	Description
`-la`	Assembled lines only
`-ld`	The `LSTOUT` directive controls if lines are written to the list file or not. Using `-ld` turns the start value for this to off.
`-le`	No macro expansions
`-lm`	Macro definitions
`-lo`	Multiline code
`-lx`	Includes cross-references
`-lN`	Do not include diagnostics

Table 13: Conditional list options (-l)

If a *`filename`* is specified, the assembler stores the output in that file.

If a *`directory`* is specified, the assembler stores the output in that directory, in a file with the extension `lst`. The filename is the same as the name of the assembled source file, unless a different name was specified with the `-o` option, in which case that name is used.

To specify the working directory, replace *`directory`* with a period (`.`).

Example 1

To generate a listing to the file `list.lst`, use:

```
iasmsh sourcefile -l list
```

Example 2

If you assemble the file `mysource.s` and want to generate a listing to a file `mysource.lst` in the working directory, you could use:

```
iasmsh mysource -l .
```

Note: You can use both `\` and `/` as directory delimiters.

To set related options, select:

Project>Options>Assembler >List

`-M` `-M*ab*`

This option sets the characters to be used as left and right quotes of each macro argument to *a* and *b* respectively.

By default, the characters are < and >. The `-M` option allows you to change the quote characters to suit an alternative convention or simply to allow a macro argument to contain < or > themselves.

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

to call a macro `print` with > as the argument.

Note: Depending on your host environment, it might be necessary to use quote marks with the macro quote characters, for example:

```
iasmsh filename -M'<>'
```

Project>Options>Assembler >Language>Macro quote characters

`--mnem_first` `--mnem_first`

The default behavior of the assembler is to treat all identifiers starting in the first column as labels.

Use this option to make mnemonics names (without a trailing colon) starting in the first column recognized as mnemonics.

Project>Options>Assembler >Language>Allow mnemonics in first column

---

`--no_fragments` `--no_fragments`

Use this option to disable section fragment handling. Normally, the toolset uses IAR proprietary information for transferring section fragment information to the linker. The

linker uses this information to remove unused code and data, and thus further minimize the size of the executable image.

To set this option, use **Project>Options>Assembler>Extra Options**

`--no_path_in_file_macros` `--no_path_in_file_macros`

Use this option to exclude the path from the return value of the predefined preprocessor symbols `__FILE__` and `__BASE_FILE__`.

This option is not available in the IDE.

`--no_system_include` `--no_system_include`

By default, the assembler automatically locates the system include files. Use this option to disable the automatic search for system include files. In this case, you might need to set up the search path by using the `-I` assembler option.

Project>Options>Assembler>Preprocessor>Ignore standard include directories

---

`--no_warnings` `--no_warnings`

By default, the assembler issues standard warning messages. Use this option to disable all warning messages.

This option is not available in the IDE.

`--no_wrap_diagnostics` `--no_wrap_diagnostics`

By default, long lines in assembler diagnostic messages are broken into several lines to make the message easier to read. Use this option to disable line wrapping of diagnostic messages.

This option is not available in the IDE.

`--only_stdout` `--only_stdout`

Causes the assembler to use `stdout` also for messages that are normally directed to `stderr`.

This option is not available in the IDE.

--output, -o

```
--output {filename|path}
-o {filename|path}
```

By default, the object code output produced by the assembler is located in a file with the same name as the source file, but with the extension `o`. Use this option to explicitly specify a different output filename for the object code output.This option sets the filename to be used for the object file.

For more syntax information, see *Setting command line assembler options*, page 17.

Project>Options>General Options>Output>Output directories>Object files

--predef_macros

```
--predef_macros {filename|directory}
```

Use this option to list the predefined symbols. When using this option, make sure to also use the same options as for the rest of your project.

If a *filename* is specified, the assembler stores the output in that file.

If a *directory* is specified, the assembler stores the output in that directory, in a file with the filename extension `predef`. To specify the working directory, replace *directory* with a period (`.`).

Note that this option requires that you specify a source file on the command line.

This option is not available in the IDE.

--preinclude

```
--preinclude includefile
```

Use this option to make the assembler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

To set this option, use:

Project>Options>Assembler>Extra Options

--preprocess

```
--preprocess=[c][n][l] {filename|directory}
```

Use this option to direct preprocessor output to a named file.

This table shows the mapping of the available preprocessor modifiers:

Command line option	Description
`--preprocess=c`	Preserve comments that otherwise are removed by the preprocessor, that is, C and C++ style comments. Assembler style comments are always preserved
`--preprocess=n`	Preprocess only
`--preprocess=l`	Generate `#line` directives

Table 14: Directing preprocessor output to file (--preprocess)

If a *`filename`* is specified, the assembler stores the output in that file.

If a *`directory`* is specified, the assembler stores the output in that directory, in a file with the extension `i`. The filename is the same as the name of the assembled source file, unless a different name was specified with the `-o` option, in which case that name is used.

To specify the working directory, replace *`directory`* with a period (`.`).

Example 1

To store the assembler output with preserved comments to the file `output.i`, use:

```
iasmsh sourcefile --preprocess=c output
```

Example 2

If you assemble the file `mysource.s` and want to store the assembler output with `#line` directives to a file `mysource.i` in the working directory, you could use:

```
iasmsh mysource --preprocess=l .
```

Note: You can use both `\` and `/` as directory delimiters.

Project>Options>Assembler >Preprocessor>Preprocessor output to file

---

`--remarks` `--remarks`

Use this option to make the assembler generate remarks, which is the least severe type of diagnostic message and which indicates a source code construct that might cause strange behavior in the generated code. By default, remarks are not generated.

See *Severity levels*, page 109, for additional information about diagnostic messages.

Project>Options>Assembler >Diagnostics>Enable remarks

-silent `--silent`

The `--silent` option causes the assembler to operate without sending any messages to the standard output stream.

By default, the assembler sends various insignificant messages via the standard output stream. Use the `--silent` option to prevent this. The assembler sends error and warning messages to the error output stream, so they are displayed regardless of this setting.

This option is not available in the IDE.

--system_include_dir `--system_include_dir` *path*

path The path to the system include files.

By default, the assembler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location.

This option is not available in the IDE.

--warnings_affect_exit_code `--warnings_affect_exit_code`

By default, the exit code is not affected by warnings, only errors produce a non-zero exit code. With this option, warnings generate a non-zero exit code.

This option is not available in the IDE.

--warnings_are_errors `--warnings_are_errors`

Use this option to make the assembler treat all warnings as errors. If the assembler encounters an error, no object code is generated.

If you want to keep some warnings, use this option in combination with the option `--diag_warning`. First make all warnings become treated as errors and then reset the ones that should still be treated as warnings, for example:

```
--diag_warning=As001
```

For additional information, see *--diag_warning*, page 25.

Project>Options>Assembler >Diagnostics>Treat all warnings as errors

Assembler operators

This chapter first describes the precedence of the assembler operators, and then summarizes the operators, classified according to their precedence. Finally, this chapter provides reference information about each operator, presented in alphabetical order.

Precedence of operators

Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The precedence numbers range from 1 (the highest precedence, that is, first evaluated) to 15 (the lowest precedence, that is, last evaluated).

These rules determine how expressions are evaluated:

- The highest precedence operators are evaluated first, then the second highest precedence operators, and so on until the lowest precedence operators are evaluated
- Operators of equal precedence are evaluated from left to right in the expression
- Parentheses (and) can be used for grouping operators and operands and for controlling the order in which the expressions are evaluated. For example, this expression evaluates to 1:

```
7/(1+(2*3))
```

Note: The precedence order in the assembler closely follows the precedence order of the Standard C++ standard for operators, where applicable.

Summary of assembler operators

The following tables give a summary of the operators, in order of precedence. Synonyms, where available, are shown in brackets after the operator name.

Note: An assembler operator can only operate on expressions that are translatable into one ELF relocation or that are absolutely soluble.

PARENTHESIS OPERATOR – 1

`()`	Parenthesis.

FUNCTION OPERATORS – 2

`BYTE1`	First byte.
`BYTE2`	Second byte.
`BYTE3`	Third byte.
`BYTE4`	Fourth byte.
`DATE`	Current date/time.
`HIGH`	High byte.
`HWRD`	High word.
`LOW`	Low byte.
`LWRD`	Low word.
`SFB`	Section begin.
`SFE`	Section end.
`SIZEOF`	Section size.
`UPPER`	Third byte.

UNARY OPERATORS – 3

`+`	Unary plus.
`BINNOT [~]`	Bitwise NOT.
`NOT [!]`	Logical NOT.
`-`	Unary minus.

MULTIPLICATIVE ARITHMETIC OPERATORS – 4

`*`	Multiplication.
`/`	Division.
`MOD [%]`	Modulo.

ADDITIVE ARITHMETIC OPERATORS – 5

`+`	Addition.
–	Subtraction.

SHIFT OPERATORS – 6

`SHL [<<]`	Logical shift left.
`SHR [>>]`	Logical shift right.

COMPARISON OPERATORS – 7

`GE [>=]`	Greater than or equal.
`GT [>]`	Greater than.
`LE [<=]`	Less than or equal.
`LT [<]`	Less than.
`UGT`	Unsigned greater than.
`ULT`	Unsigned less than.

EQUIVALENCE OPERATORS – 8

`EQ [=] [==]`	Equal.
`NE [<>] [!=]`	Not equal.

LOGICAL OPERATORS – 9-14

`BINAND [&]`	Bitwise AND (9).
`BINXOR [^]`	Bitwise exclusive OR (10).
`BINOR [|]`	Bitwise OR (11).

`AND [&&]`	Logical AND (12).
`XOR`	Logical exclusive OR (13).
`OR [||]`	Logical OR (14).

CONDITIONAL OPERATOR – 15

`?:`	Conditional operator.

Description of assembler operators

The following sections give full descriptions of each assembler operator. The number within parentheses specifies the priority of the operator

---

`()` Parenthesis (1).

`(` and `)` group expressions to be evaluated separately, overriding the default precedence order.

Example

```
1+2*3 → 7
(1+2)*3 → 9
```

`*` Multiplication (4).

* produces the product of its two operands. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

Example

```
2*2 → 4
-2*2 → -4
```

`+` Unary plus (3).

Unary plus operator.

Example

```
+3 → 3
3*+2 → 6
```

+ Addition (5).

The + addition operator produces the sum of the two operands which surround it. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

Example

```
92+19 → 111
-2+2 → 0
-2+-2 → -4
```

– Unary minus (3).

The unary minus operator performs arithmetic negation on its operand.

The operand is interpreted as a 32-bit signed integer and the result of the operator is the two's complement negation of that integer.

Example

```
-3 → -3
3*-2 → -6
4--5 → 9
```

– Subtraction (5).

The subtraction operator produces the difference when the right operand is taken away from the left operand. The operands are taken as signed 32-bit integers and the result is also signed 32-bit integer.

Example

```
92-19 → 73
-2-2 → -4
-2--2 → 0
```

/ Division (4).

/ produces the integer quotient of the left operand divided by the right operand. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

Example

```
9/2 → 4
-12/3 → -4
9/2*6 → 24
```

`?:` Conditional operator (15).

The result of this operator is the first *`expr`* if *`condition`* evaluates to true and the second *`expr`* if *`condition`* evaluates to false.

Note: The question mark and a following label must be separated by space or a tab, otherwise the ? is considered the first character of the label.

Syntax

`condition` `?` *`expr`* `:` *`expr`*

Example

```
5 ? 6 : 7 →6
0 ? 6 : 7 →7
```

`AND [&&]` Logical AND (12).

Use `AND` to perform logical AND between its two integer operands. If both operands are non-zero the result is 1 (true), otherwise it is 0 (false).

Example

```
1010B AND 0011B → 1
1010B AND 0101B → 1
1010B AND 0000B → 0
```

`BINAND [&]` Bitwise AND (9).

Use `BINAND` to perform bitwise AND between the integer operands. Each bit in the 32-bit result is the logical AND of the corresponding bits in the operands.

Example

```
1010B BINAND 0011B → 0010B
1010B BINAND 0101B → 0000B
1010B BINAND 0000B → 0000B
```

`BINNOT [~]` Bitwise NOT (3).

Use `BINNOT` to perform bitwise NOT on its operand. Each bit in the 32-bit result is the complement of the corresponding bit in the operand.

Example

```
BINNOT 1010B → 11111111111111111111111111110101B
```

`BINOR [|]` Bitwise OR (11).

Use `BINOR` to perform bitwise OR on its operands. Each bit in the 32-bit result is the inclusive OR of the corresponding bits in the operands.

Example

```
1010B BINOR 0101B → 1111B
1010B BINOR 0000B → 1010B
```

`BINXOR [^]` Bitwise exclusive OR (10).

Use `BINXOR` to perform bitwise XOR on its operands. Each bit in the 32-bit result is the exclusive OR of the corresponding bits in the operands.

Example

```
1010B BINXOR 0101B → 1111B
1010B BINXOR 0011B → 1001B
```

`BYTE1` First byte (2).

`BYTE1` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the low byte (bits 7 to 0) of the operand.

Example

```
BYTE1 0x12345678 → 0x78
```

`BYTE2` Second byte (2).

`BYTE2` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the middle-low byte (bits 15 to 8) of the operand.

Example

```
BYTE2 0x12345678 → 0x56
```

`BYTE3` Third byte (2).

`BYTE3` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the middle-high byte (bits 23 to 16) of the operand.

Example

```
BYTE3 0x12345678 → 0x34
```

`BYTE4` Fourth byte (2).

`BYTE4` takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the high byte (bits 31 to 24) of the operand.

Example

```
BYTE4 0x12345678 → 0x12
```

`DATE` Current date/time (2).

Use the `DATE` operator to specify when the current assembly began.

The `DATE` operator takes an absolute argument (expression) and returns:

`DATE 1`	Current second (0–59)
`DATE 2`	Current minute (0–59)
`DATE 3`	Current hour (0–23)
`DATE 4`	Current day (1–31)
`DATE 5`	Current month (1–12)
`DATE 6`	Current year `MOD 100` (1998 →98, 2000 →00, 2002 →02)

Example

To assemble the date of assembly:

```
today: DC8 DATE 5, DATE 4, DATE 3
```

`EQ [=] [==]` Equal (8).

`=` evaluates to 1 (true) if its two operands are identical in value, or to 0 (false) if its two operands are not identical in value.

Example

```
1 = 2 → 0
2 == 2 → 1
'ABC' = 'ABCD' → 0
```

`GE [>=]` Greater than or equal (7).

`>=` evaluates to 1 (true) if the left operand is equal to or has a higher numeric value than the right operand, otherwise it is 0 (false).

Example

```
1 >= 2 → 0
2 >= 1 → 1
1 >= 1 → 1
```

`GT [>]` Greater than (7).

`>` evaluates to 1 (true) if the left operand has a higher numeric value than the right operand, otherwise it is 0 (false).

Example

```
-1 > 1 → 0
2 > 1 → 1
1 > 1 → 0
```

`HIGH` High byte (2).

`HIGH` takes a single operand to its right which is interpreted as an unsigned, 16-bit integer value. The result is the unsigned 8-bit integer value of the higher order byte of the operand.

Example

```
HIGH 0xABCD → 0xAB
```

`HWRD` High word (2).

`HWRD` takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the high word (bits 31 to 16) of the operand.

Example

```
HWRD 0x12345678 → 0x1234
```

`LE [<=]` Less than or equal (7).

<= evaluates to 1 (true) if the left operand has a lower or equal numeric value to the right operand, otherwise it is 0 (false).

Example

```
1 <= 2 → 1
2 <= 1 → 0
1 <= 1 → 1
```

`LOW` Low byte (2).

`LOW` takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the unsigned, 8-bit integer value of the lower order byte of the operand.

Example

```
LOW 0xABCD → 0xCD
```

`LT [<]` Less than (7).

< evaluates to 1 (true) if the left operand has a lower numeric value than the right operand, otherwise it is 0 (false).

Example

```
-1 < 2 → 1
2 < 1 → 0
2 < 2 → 0
```

`LWRD` Low word (2).

`LWRD` takes a single operand, which is interpreted as an unsigned, 32-bit integer value. The result is the low word (bits 15 to 0) of the operand.

Example

```
LWRD 0x12345678 → 0x5678
```

`MOD [%]` Modulo (4).

`MOD` produces the remainder from the integer division of the left operand by the right operand. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

`X MOD Y` is equivalent to `X-Y*(X/Y)` using integer division.

Example

```
2 MOD 2 → 0
12 MOD 7 → 5
3 MOD 2 → 1
```

`NE [<>] [!=]` Not equal (8).

`<>` evaluates to 0 (false) if its two operands are identical in value or to 1 (true) if its two operands are not identical in value.

Example

```
1 <> 2 → 1
2 <> 2 → 0
'A' <> 'B' → 1
```

`NOT [!]` Logical NOT (3).

Use `NOT` to negate a logical argument.

Example

```
NOT 0101B → 0
NOT 0000B → 1
```

`OR [||]` Logical OR (14).

Use `OR` to perform a logical OR between two integer operands.

Example

```
1010B OR 0000B → 1
0000B OR 0000B → 0
```

SFB Section begin (2).

SFB accepts a single operand to its right. The operand must be the name of a relocatable section. The operator evaluates to the absolute address of the first byte of that section. This evaluation occurs at link time.

Syntax

SFB(*section* [{+|-}*offset*])

Parameters

section	The name of a relocatable section, which must be defined before SFB is used.
offset	An optional offset from the start address. The parentheses are optional if *offset* is omitted.

Example

```
            name    sectionBegin
            section MYCODE:CODE  ; Forward declaration of MYCODE.
            section SEGTAB:CONST
start       dc16    sfb(MYCODE)
            end
```

Even if this code is linked with many other modules, start is still set to the address of the first byte of the section.

SFE Section end (2).

SFE accepts a single operand to its right. The operand must be the name of a relocatable section. The operator evaluates to the section start address plus the section size. This evaluation occurs at link time.

Syntax

SFE (*section* [{+ | -} *offset*])

Parameters

section	The name of a relocatable section, which must be defined before SFE is used.
offset	An optional offset from the start address. The parentheses are optional if *offset* is omitted.

Example

```
      name    demo
      section segtab:CONST
end:  dc16    SFE(mycode)
```

Even if this code is linked with many other modules, `end` is still set to the first byte after that section (`mycode`).

The size of the section `MY_SECTION` can be calculated as:

```
SFE(MY_SECTION)-SFB(MY_SECTION)
```

`SHL [<<]` Logical shift left (6).

Use `SHL` to shift the left operand, which is always treated as `unsigned`, to the left. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.

Example

```
00011100B SHL 3 → 11100000B
0000011111111111B SHL 5 → 1111111111100000B
14 SHL 1 → 28
```

`SHR [>>]` Logical shift right (6).

Use `SHR` to shift the left operand, which is always treated as `unsigned`, to the right. The number of bits to shift is specified by the right operand, interpreted as an integer value between 0 and 32.

Example

```
01110000B SHR 3 → 00001110B
1111111111111111B SHR 20 → 0
14 SHR 1 → 7
```

`SIZEOF` Section size (2).

`SIZEOF` generates `SFE-SFB` for its argument, which should be the name of a relocatable section; that is, it calculates the size in bytes of a section. This is done when modules are linked together.

Syntax

`SIZEOF (`*`section`*`)`

Parameters

section The name of a relocatable section, which must be defined before SIZEOF is used.

Example

This code sets size to the size of the section mycode.

```
            name    table
            section MYCODE:CODE   ; Forward declaration of MYCODE
            section SEGTAB:CONST(2)
            data
size        dc32    sizeof(MYCODE)

            section MYCODE:CODE(2)
            code
            nop                   ; Placeholder for application.
            end
```

UGT Unsigned greater than (7).

UGT evaluates to 1 (true) if the left operand has a larger value than the right operand, otherwise it is 0 (false). The operation treats its operands as unsigned values.

Example

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT Unsigned less than (7).

ULT evaluates to 1 (true) if the left operand has a smaller value than the right operand, otherwise it is 0 (false). The operation treats the operands as unsigned values.

Example

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

UPPER Third byte (2).

UPPER takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the middle-high byte (bits 23 to 16) of the operand.

Example

```
UPPER 0x12345678 → 0x34
```

XOR Logical exclusive OR (13).

XOR evaluates to 1 (true) if either the left operand or the right operand is non-zero, but to 0 (false) if both operands are zero or both are non-zero. Use XOR to perform logical XOR on its two operands.

Example

```
0101B XOR 1010B → 0
0101B XOR 0000B → 1
```

Assembler directives

This chapter gives an alphabetical summary of the assembler directives and provides detailed reference information for each category of directives.

Summary of assembler directives

The assembler directives are classified into these groups according to their function:

- *Module control directives*, page 55
- *Symbol control directives*, page 57
- *Mode control directives*, page 59
- *Section control directives*, page 60
- *Value assignment directives*, page 64
- *Conditional assembly directives*, page 66
- *Macro processing directives*, page 69
- *Listing control directives*, page 77
- *C-style preprocessor directives*, page 81
- *Data definition or allocation directives*, page 86
- *Assembler control directives*, page 89
- *Call frame information directives*, page 92.

This table gives a summary of all the assembler directives.

Directive	Description	Section
`_args`	Is set to number of arguments passed to macro.	Macro processing
`#define`	Assigns a value to a label.	C-style preprocessor
`#elif`	Introduces a new condition in a `#if`…`#endif` block.	C-style preprocessor
`#else`	Assembles instructions if a condition is false.	C-style preprocessor
`#endif`	Ends a `#if`, `#ifdef`, or `#ifndef` block.	C-style preprocessor
`#error`	Generates an error.	C-style preprocessor
`#if`	Assembles instructions if a condition is true.	C-style preprocessor
`#ifdef`	Assembles instructions if a symbol is defined.	C-style preprocessor
`#ifndef`	Assembles instructions if a symbol is undefined.	C-style preprocessor

Table 15: Assembler directives summary

Directive	Description	Section
`#include`	Includes a file.	C-style preprocessor
`#line`	Changes the line numbers.	C-style preprocessor
`#pragma`	Controls extension features.	C-style preprocessor
`#undef`	Undefines a label.	C-style preprocessor
`/*comment*/`	C-style comment delimiter.	Assembler control
`//`	C++ style comment delimiter.	Assembler control
`=`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the program location counter by inserting zero-filled bytes.	Section control
`ALIGNRAM`	Aligns the program location counter.	Section control
`ASSIGN`	Assigns a temporary value.	Value assignment
`CASEOFF`	Disables case sensitivity.	Assembler control
`CASEON`	Enables case sensitivity.	Assembler control
`CFI`	Specifies call frame information.	Call frame information
`CODE`	Subsequent instructions are disassembled as code.	Mode control
`DATA`	Subsequent instructions are disassembled as 8-bit data.	Mode control
`DATA8`	Subsequent instructions are disassembled as 8-bit data.	Mode control
`DATA16`	Subsequent instructions are disassembled as 16-bit data.	Mode control
`DATA32`	Subsequent instructions are disassembled as 32-bit data.	Mode control
`DATA64`	Subsequent instructions are disassembled as 64-bit data.	Mode control
`DC8`	Generates 8-bit constants, including strings.	Data definition or allocation
`DC16`	Generates 16-bit constants.	Data definition or allocation
`DC24`	Generates 24-bit constants.	Data definition or allocation
`DC32`	Generates 32-bit constants.	Data definition or allocation

Table 15: Assembler directives summary (Continued)

Directive	Description	Section
DC64	Generates 64-bit constants.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DF32	Generates 32-bit floating-point constants.	Data definition or allocation
DF64	Generates 64-bit floating-point constants.	Data definition or allocation
DQ15	Generates 16-bit fractional constants.	Data definition or allocation
DQ31	Generates 32-bit fractional constants.	Data definition or allocation
DS8	Allocates space for 8-bit integers.	Data definition or allocation
DS16	Allocates space for 16-bit integers.	Data definition or allocation
DS24	Allocates space for 24-bit integers.	Data definition or allocation
DS32	Allocates space for 32-bit integers.	Data definition or allocation
DS64	Allocates space for 64-bit integers.	Data definition or allocation
ELSE	Assembles instructions if a condition is false.	Conditional assembly
ELSEIF	Specifies a new condition in an IF...ENDIF block.	Conditional assembly
END	Ends the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing
ENDR	Ends a repeat structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Section control
EXITM	Exits prematurely from a macro.	Macro processing
EXTERN	Imports an external symbol.	Symbol control
IF	Assembles instructions if a condition is true.	Conditional assembly
IMPORT	Imports an external symbol.	Symbol control
LIBRARY	Begins a module; an alias for PROGRAM and NAME.	Module control

Table 15: Assembler directives summary (Continued)

Directive	Description	Section
LOCAL	Creates symbols local to a macro.	Macro processing
LSTCND	Controls conditional assembler listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembler-listing output.	Listing control
LSTPAG	Retained for backward compatibility reasons; recognized but ignored.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
LSTXRF	Generates a cross-reference table.	Listing control
MACRO	Defines a macro.	Macro processing
MODULE	Begins a module; an alias for PROGRAM and NAME.	Module control
NAME	Begins a program module.	Module control
ODD	Aligns the program location counter to an odd address.	Section control
OVERLAY	Recognized but ignored.	Symbol control
POOL	Specifies where to place constant tables.	Assembler control
PROGRAM	Begins a module.	Module control
PUBLIC	Exports symbols to other modules.	Symbol control
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.	Symbol control
RADIX	Sets the default base.	Assembler control
REPT	Assembles instructions a specified number of times.	Macro processing
REPTC	Repeats and substitutes characters.	Macro processing
REPTI	Repeats and substitutes strings.	Macro processing
REQUIRE	Forces a symbol to be referenced.	Symbol control
RSEG	Begins a section.	Section control
RTMODEL	Declares runtime model attributes.	Module control
SECTION	Begins a section.	Section control
SECTION_TYPE	Sets ELF type and flags for a section.	Section control
SET	Assigns a temporary value.	Value assignment
VAR	Assigns a temporary value.	Value assignment

Table 15: Assembler directives summary (Continued)

Module control directives

Module control directives are used for marking the beginning and end of source program modules, and for assigning names to them. See *Expression restrictions*, page 13, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
END	Ends the assembly of the last module in a file.	Only locally defined labels or integer constants
NAME	Begins a module.	No external references Absolute
PROGRAM	Begins a module.	No external references Absolute
RTMODEL	Declares runtime model attributes.	Not applicable

Table 16: Module control directives

SYNTAX

```
END
NAME symbol
PROGRAM symbol
RTMODEL key, value
```

PARAMETERS

key	A text string specifying the key.
symbol	Name assigned to module.
value	A text string specifying the value.

DESCRIPTIONS

Beginning a module

Use any of the directives NAME or PROGRAMHistory to begin an ELF module, and to assign a name.

A module is included in the linked application, even if other modules do not reference them. For more information about how modules are included in the linked application, read about the linking process in the *IAR C/C++ Development Guide for SH*.

Note: There can be only one module in a file.

Terminating the source file

Use END to indicate the end of the source file. Any lines after the END directive are ignored. The END directive also ends the module in the file.

Declaring runtime model attributes

Use RTMODEL to enforce consistency between modules. All modules that are linked together and define the same runtime attribute key must have the same value for the corresponding key value, or the special value *. Using the special value * is equivalent to not defining the attribute at all. It can however be useful to explicitly state that the module can handle any runtime model.

A module can have several runtime model definitions.

Note: The compiler runtime model attributes start with double underscores. In order to avoid confusion, this style must not be used in the user-defined assembler attributes.

If you are writing assembler routines for use with C or C++ code, and you want to control the module consistency, refer to the *IAR C/C++ Development Guide for SH.*

Examples

The following examples defines three modules in one source file each, where:

- MOD_1 and MOD_2 *cannot* be linked together since they have different values for runtime model CAN.
- MOD_1 and MOD_3 *can* be linked together since they have the same definition of runtime model RTOS and no conflict in the definition of CAN.
- MOD_2 and MOD_3 *can* be linked together since they have no runtime model conflicts. The value * matches any runtime model value.

Assembler source file f1.s:

```
module  mod_1
rtmodel "CAN",  "ISO11519"
rtmodel "RTOS", "PowerPac"
; ...
end
```

Assembler source file f2.s:

```
module  mod_2
rtmodel "CAN",  "ISO11898"
rtmodel "RTOS", "*"
; ...
end
```

Assembler source file `f3.s`:

```
module  mod_3
rtmodel "RTOS", "PowerPac"
; ...
end
```

Symbol control directives

These directives control how symbols are shared between modules.

Directive	Description
EXTERN, IMPORT	Imports an external symbol.
OVERLAY	Recognized but ignored.
PUBLIC	Exports symbols to other modules.
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.
REQUIRE	Forces a symbol to be referenced.

Table 17: Symbol control directives

SYNTAX

```
EXTERN symbol [,symbol] …
IMPORT symbol [,symbol] …
PUBLIC symbol [,symbol] …
PUBWEAK symbol [,symbol] …
REQUIRE symbol
```

PARAMETERS

label	Label to be used as an alias for a C/C++ symbol.
symbol	Symbol to be imported or exported.

DESCRIPTIONS

Exporting symbols to other modules

Use PUBLIC to make one or more symbols available to other modules. Symbols defined PUBLIC can be relocatable or absolute, and can also be used in expressions (with the same rules as for other symbols).

The PUBLIC directive always exports full 32-bit values, which makes it feasible to use global 32-bit constants also in assemblers for 8-bit and 16-bit processors. With the LOW,

HIGH, >>, and << operators, any part of such a constant can be loaded in an 8-bit or 16-bit register or word.

There can be any number of PUBLIC-defined symbols in a module.

Exporting symbols with multiple definitions to other modules

PUBWEAK is similar to PUBLIC except that it allows the same symbol to be defined several times. Only one of those definitions is used by ILINK. If a module containing a PUBLIC definition of a symbol is linked with one or more modules containing PUBWEAK definitions of the same symbol, ILINK uses the PUBLIC definition.

A section cannot contain both a public symbol and a pubweak symbols.

Note: Library modules are only linked if a reference to a symbol in that module is made, and that symbol was not already linked. During the module selection phase, no distinction is made between PUBLIC and PUBWEAK definitions. This means that to ensure that the module containing the PUBLIC definition is selected, you should link it before the other modules, or make sure that a reference is made to some other PUBLIC symbol in that module.

Importing symbols

Use EXTERN or IMPORT to import an untyped external symbol.

The REQUIRE directive marks a symbol as referenced. This is useful if the section containing the symbol must be loaded for the code containing the reference to work, but the dependence is not otherwise evident.

EXAMPLES

The following example defines a subroutine to print an error message, and exports the entry address err so that it can be called from other modules.

Because the message is enclosed in double quotes, the string will be followed by a zero byte.

It defines print as an external routine; the address is resolved at link time.

```
            name    errorMessage
            extern  print
            public  err
            section __DEFAULT_CODE_SECTION__:CODE(2)

err         mova    error,R0        ; Load address of error
                                    ; string in R0.
            mov.l   ct_print,R1     ; Move print function address
                                    ; to R1
            jsr     @R1
```

```
            nop
            rts/n

            section __DEFAULT_CODE_SECTION__:CODE(2)
            DATA
ct_print    dc32    print
error       dc8     "** Error **"
            end
```

Mode control directives

These directives provide control over the disassembly mode:

Directive	Description
CODE	Subsequent instructions are disassembled as code.
DATA, DATA8	Subsequent instructions are disassembled as 8-bit data.
DATA16	Subsequent instructions are disassembled as 16-bit data.
DATA32	Subsequent instructions are disassembled as 32-bit data.
DATA64	Subsequent instructions are disassembled as 64-bit data.

Table 18: Mode control directives

SYNTAX

```
CODE
DATA
DATA8
DATA16
DATA32
DATA64
```

DESCRIPTION

The CODE and DATA directives set the disassembly mode for code and data sections. This information is used by C-SPY and IAR ELF Dumper.

The CODE or DATA directives can be used for:

- Starting a code/data producing section fragment (SECTION) that actually generates bytes that end up in the image, either code or data
- Changing the disassembly mode in the middle of a section fragment.

The directive should come after the section fragment start (for example after the SECTION directive) and immediately precede any code-generating part (instructions or DC declarations).

You do not need the CODE or DATA directives for declaring sections, extern labels etc, and not when you declare RAM space.

EXAMPLES

In this example, the disassembly mode changes several times to accommodate different types of data.

```
        name    codedata
        extern  printStr
        public  printDate
        section __DEFAULT_CODE_SECTION__:CODE(2)

        code                    ; Disassembled as code
printDate:  mova    a_date,R0   ; Load address of date
                                ; string in R0.
        mov.l   ct_printStr,R1  ; Move printStr function address
                                ; to R1
        jsr     @R1             ; Call string output routine.
        nop
        rts/n
        data32                  ; Disassembled as 32-bit data
ct_printStr:
        dc32    printStr
        data8                   ; Disassembled as 8-bit data
a_date:
        dc8     __DATE__        ; String representing the
                                ; date of assembly.
        end
```

Section control directives

The section directives control how code and data are located. See *Expression restrictions*, page 13, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
ALIGN	Aligns the program location counter by inserting zero-filled bytes.	No external references Absolute
ALIGNRAM	Aligns the program location counter.	No external references Absolute
EVEN	Aligns the program counter to an even address.	No external references Absolute

Table 19: Section control directives

Directive	Description	Expression restrictions
ODD	Aligns the program counter to an odd address.	No external references Absolute
RSEG	Begins a section; alias to SECTION.	No external references Absolute
SECTION	Begins an ELF section.	No external references Absolute
SECTION_TYPE	Sets ELF type and flags for a section.	

Table 19: Section control directives (Continued)

SYNTAX

```
ALIGN align [,value]
ALIGNRAM align
EVEN [value]
ODD [value]
RSEG section :type [flag] [(align)]
SECTION section :type [flag] [(align)]
SECTION_TYPE type-expr {,flags-expr}
```

PARAMETERS

align
The power of two to which the address should be aligned, in most cases in the range 0 to 30.
The default align value is 0, except for code sections where the default is 1.

flag
NOROOT, ROOT
NOROOT means that the section fragment is discarded by the linker if no symbols in this section fragment are referred to. Normally, all section fragments except startup code and interrupt vectors should set this flag. The default mode is ROOT which indicates that the section fragment must not be discarded.

REORDER, NOREORDER
REORDER starts a new section with the given name. The default mode is NOREORDER which starts a new fragment in the section with the given name, or a new section if no such section exists.

section
The name of the section.

type
The memory type, which can be either CODE, CONST, or DATA.

value	Byte value used for padding, default is zero.
type-expr	A constant expression identifying the ELF type of the section.
flags-expr	A constant expression identifying the ELF flags of the section.

DESCRIPTIONS

Beginning a relocatable section

Use SECTION (or RSEG) to start a new section. The assembler maintains separate location counters (initially set to zero) for all sections, which makes it possible to switch sections and mode anytime without having to save the current program location counter.

Note: The first instance of a SECTION or RSEG directive must not be preceded by any code generating directives, such as DC or DS directives, or by any assembler instructions.

To set the ELF type, and possibly the ELF flags for the newly created section, use SECTION_TYPE. By default, the values of the flags are zero. For information about valid values, refer to the ELF documentation.

Aligning a section

Use ALIGN to align the program location counter to a specified address boundary. The expression gives the power of two to which the program counter should be aligned and the permitted range is 0 to 8.

The alignment is made relative to the section start; normally this means that the section alignment must be at least as large as that of the alignment directive to give the desired result.

ALIGN aligns by inserting zero/filled bytes, up to a maximum of 255. The EVEN directive aligns the program counter to an even address (which is equivalent to ALIGN 1) and the ODD directive aligns the program location counter to an odd address. The byte value for padding must be within the range 0 to 255.

Use ALIGNRAM to align the program location counter by incrementing it; no data is generated. The expression can be within the range 0 to 30.

EXAMPLES

Beginning a relocatable section

In the following example, the data following the first SECTION directive is placed in a section called table.

The code following the second SECTION directive is placed in a relocatable section called CODE:

```
            module  calculate
            extern  operator
            extern  addOperator, subOperator

            section TABLE:CONST(8)
            DATA
operatorTable:
            dc8     addOperator, subOperator

            section __DEFAULT_CODE_SECTION__:CODE(2)
            CODE
calculate   mov.l   ct_operatorTable,r0
            mov.l   ct_operator,r1
            mov.b   @r1,r2      ;Get operator to r2
            mov.b   @r0+,r3     ;Get addOperator to r3
            cmp/eq  r2,r3
            bt      add
            mov.b   @r0+,r3     ;Get addOperator to r3
            cmp/eq  r2,r3
            bt      sub
            ;...
            rts
            nop
            section __DEFAULT_CODE_SECTION__:CODE(2)
ct_operator DC32    operator
ct_operatorTable:
            DC32    operatorTable

            section __DEFAULT_CODE_SECTION__:CODE(2)
add         ;...
            rts
            nop
sub         ;...
            rts
            nop

            end
```

Aligning a section

This example starts a section, moves to an even address, and adds some data. It then aligns to a 64-byte boundary before creating a 64-byte table.

```
            name    alignment
            section DATA:DATA ; Start a relocatable data section.
            DATA
            even              ; Ensure it is on an even boundary.
target      dc16    1         ; target and best will be on an
best        dc16    1         ; even boundary.
            align   6         ; Now, align to a 64-byte boundary,
results     ds8     64        ; and create a 64-byte table.
            end
```

Value assignment directives

These directives are used for assigning values to symbols.

Directive	Description
=, EQU	Assigns a permanent value local to a module.
ASSIGN, SET, VAR	Assigns a temporary value.
DEFINE	Defines a file-wide value.

Table 20: Value assignment directives

SYNTAX

```
label = expr
label ASSIGN expr
label DEFINE const_expr
label EQU expr
label SET expr
label VAR expr
```

PARAMETERS

const_expr	Constant value assigned to symbol.
expr	Value assigned to symbol or value to be tested.
label	Symbol to be defined.

DESCRIPTIONS

Defining a temporary value

Use ASSIGN, SET, or VAR to define a symbol that might be redefined, such as for use with macro variables. Symbols defined with ASSIGN, SET, or VAR cannot be declared PUBLIC.

Defining a permanent local value

Use EQU or = to create a local symbol that denotes a number or offset. The symbol is only valid in the module in which it was defined, but can be made available to other modules with a PUBLIC directive (but not with a PUBWEAK directive).

Use EXTERN to import symbols from other modules.

Defining a permanent global value

Use DEFINE to define symbols that should be known to the module containing the directive. After the DEFINE directive, the symbol is known.

A symbol which was given a value with DEFINE can be made available to modules in other files with the PUBLIC directive. Symbols defined with DEFINE cannot be redefined within the same file. Also, the expression assigned to the defined symbol must be constant.

EXAMPLES

Redefining a symbol

This example uses SET to redefine the symbol cons in a loop to generate a table of the first 8 powers of 3:

```
            name    table
cons        set     1

; Generate table of powers of 3.
cr_tabl     macro   times
            dc32    cons
cons        set     cons * 3
            if      times > 1
            cr_tabl times - 1
            endif
            endm

            section `.text`:CODE(2)
table       cr_tabl 4
            end
```

It generates this code:

```
 9                                      name    table
10    000001              cons         set     1
11
12                        ; Generate table of powers of 3.
20
21    000000                           section
                                       __DEFAULT_CODE_SECTION__:CODE(2)
22    000000              table        cr_tabl 4
22.1  000000 00000001                  dc32    cons
22.2  000003              cons         set     cons * 3
22.3  000004                           if      4 > 1
22.4  000004                           cr_tabl 4 - 1
22.5  000004 00000003                  dc32    cons
22.6  000009              cons         set     cons * 3
22.7  000008                           if      4 - 1 > 1
22.8  000008                           cr_tabl 4 - 1 - 1
22.9  000008 00000009                  dc32    cons
22.10 00001B              cons         set     cons * 3
22.11 00000C                           if      4 - 1 - 1 > 1
22.12 00000C                           cr_tabl 4 - 1 - 1 - 1
22.13 00000C 0000001B                  dc32    cons
22.14 000051              cons         set     cons * 3
22.15 000010                           if      4 - 1 - 1 - 1 > 1
22.16 000010                           endif
22.17 000010                           endif
22.18 000010                           endif
22.19 000010                           endif
23    000010                           end
```

Conditional assembly directives

These directives provide logical control over the selective assembly of source code. See *Expression restrictions*, page 13, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
ELSE	Assembles instructions if a condition is false.	
ELSEIF	Specifies a new condition in an IF…ENDIF block.	No forward references No external references Absolute Fixed

Table 21: Conditional assembly directives

Directive	Description	Expression restrictions
ENDIF	Ends an IF block.	
IF	Assembles instructions if a condition is true.	No forward references No external references Absolute Fixed

Table 21: Conditional assembly directives (Continued)

SYNTAX

```
ELSE
ELSEIF condition
ENDIF
IF condition
```

PARAMETERS

condition	One of these:	
	An absolute expression	The expression must not contain forward or external references, and any non-zero value is considered as true.
	string1==string2	The condition is true if *string1* and *string2* have the same length and contents.
	string1!=string2	The condition is true if *string1* and *string2* have different length or contents.

DESCRIPTIONS

Use the IF, ELSE, and ENDIF directives to control the assembly process at assembly time. If the condition following the IF directive is not true, the subsequent instructions do not generate any code (that is, it is not assembled or syntax checked) until an ELSE or ENDIF directive is found.

Use ELSEIF to introduce a new condition after an IF directive. Conditional assembly directives can be used anywhere in an assembly, but have their greatest use in conjunction with macro processing.

All assembler directives (except for END) as well as the inclusion of files can be disabled by the conditional directives. Each IF directive must be terminated by an ENDIF

directive. The ELSE directive is optional, and if used, it must be inside an IF...ENDIF block. IF...ENDIF and IF...ELSE...ENDIF blocks can be nested to any level.

EXAMPLES

This example uses a macro to add a constant to a direct page memory location:

```
; If the second argument to the addMem macro fits into a byte,
; 20bits word, or long, it will generate code for add byte, 20bit
; word or a long.
addMem      macro   loc,val         ; loc is a direct page memory
                                    ; location, and val is an
                                    ; 32-bit value to add to that
                                    ; location.
            if      val = 0
                                    ; Do nothing.
            elseif  val < 0x80 AND val > -0x81
            mov.l   #loc,r0
            mov.l   @r0,r1
            add     #val,r1
            mov.l   r1,@r0
            elseif  val < 0x80000 AND val > -0x80001
            mov.l   #loc,r0
            mov.l   @r0,r1
            movi20  #val,r2
            add     r2,r1
            mov.l   r1,@r0
            else
            mov.l   #loc,r0
            mov.l   @r0,r1
            mov.l   #val,r2
            add     r2,r1
            mov.l   r1,@r0
            endif
            endm


            module  addWithMacro
            section __DEFAULT_CODE_SECTION__:CODE(2)

addSome     addMem  0xa0,0          ; Add 0 to memory location
                                    ; 0xa0.
            addMem  0xa0,1          ; Add 1 to the same address.
            addMem  0xa0,1234       ; Add 1234 to the same
                                    ; address.
            addMem  0xa0,1234567    ; Add 1234567 to the same
                                    ; address.
            rts/n
```

```
pool
end
```

Macro processing directives

These directives allow user macros to be defined. See *Expression restrictions*, page 13, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
_args	Is set to number of arguments passed to macro.	
ENDM	Ends a macro definition.	
ENDR	Ends a repeat structure.	
EXITM	Exits prematurely from a macro.	
LOCAL	Creates symbols local to a macro.	
MACRO	Defines a macro.	
REPT	Assembles instructions a specified number of times.	No forward references No external references Absolute Fixed
REPTC	Repeats and substitutes characters.	
REPTI	Repeats and substitutes text.	

Table 22: Macro processing directives

SYNTAX

```
_args
ENDM
ENDR
EXITM
LOCAL symbol [,symbol] …
name MACRO [argument] [,argument] …
REPT expr
REPTC formal,actual
REPTI formal,actual [,actual] …
```

PARAMETERS

actual	A string to be substituted.
argument	A symbolic argument name.

expr	An expression.
formal	An argument into which each character of *actual* (REPTC) or each *actual* (REPTI) is substituted.
name	The name of the macro.
symbol	A symbol to be local to the macro.

DESCRIPTIONS

A macro is a user-defined symbol that represents a block of one or more assembler source lines. Once you have defined a macro, you can use it in your program like an assembler directive or assembler mnemonic.

When the assembler encounters a macro, it looks up the macro's definition, and inserts the lines that the macro represents as if they were included in the source file at that position.

Macros perform simple text substitution effectively, and you can control what they substitute by supplying parameters to them.

Defining a macro

You define a macro with the statement:

name MACRO [*argument*] [,*argument*] …

Here *name* is the name you are going to use for the macro, and *argument* is an argument for values that you want to pass to the macro when it is expanded.

For example, you could define a macro errMac as follows:

```
errMac      macro   text
            extern  abort
            mov.l   #abort,r0
            jsr     @r0
            nop
            dc8     text,0
            endm
```

This macro uses a parameter text to set up an error message for a routine abort. You would call the macro with a statement such as:

```
            errMac  'Disk not ready'
```

The assembler expands this to:

```
            extern  abort
            mov.l   #abort,r0
            jsr     @r0
```

```
            nop
            dc8     'Disk not ready',0
```

If you omit a list of one or more arguments, the arguments you supply when calling the macro are called `\1` to `\9` and `\A` to `\Z`.

The previous example could therefore be written as follows:

```
errMac      macro   text
            extern  abort
            mov.l   #abort,r0
            jsr     @r0
            nop
            dc8     \1,0
            endm
```

Use the `EXITM` directive to generate a premature exit from a macro.

`EXITM` is not allowed inside `REPT`...`ENDR`, `REPTC`...`ENDR`, or `REPTI`...`ENDR` blocks.

Use `LOCAL` to create symbols local to a macro. The `LOCAL` directive must be used before the symbol is used.

Each time that a macro is expanded, new instances of local symbols are created by the `LOCAL` directive. Therefore, it is legal to use local symbols in recursive macros.

Note: It is illegal to *redefine* a macro.

Passing special characters

Macro arguments that include commas or white space can be forced to be interpreted as one argument by using the matching quote characters `<` and `>` in the macro call.

For example:

```
movMac      macro   op
            mov.l   op
            endm
```

The macro can be called using the macro quote characters:

```
            movMac  <@(R0,R1),R2>
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 29.

Predefined macro symbols

The symbol `_args` is set to the number of arguments passed to the macro. This example shows how `_args` can be used:

```
fill        macro
            if      _args == 2
            rept    \2
            dc8     \1
            endr
            else
            dc8     \1
            endif
            endm

            section __DEFAULT_CODE_SECTION__:CODE
            fill    3
            fill    4, 3
            end
```

It generates this code:

```
     19                     module  fill
     20    000000           section __DEFAULT_CODE_SECTION__:CODE
     21    000000           fill    3
     21.1  000000           if      _args == 2
     21.2  000000           else
     21.3  000000 03        dc8     3
     21.4  000001           endif
     22    000001           fill    4, 3
     22.1  000001           if      _args == 2
     22.2  000001           rept    3
     22.3  000001 04        dc8     4
     22.4  000002 04        dc8     4
     22.5  000003 04        dc8     4
     22.6  000004           endr
     22.7  000004           else
     22.8  000004           endif
     23    000004           end
```

How macros are processed

The macro process consists of three distinct phases:

1 The assembler scans and saves macro definitions. The text between `MACRO` and `ENDM` is saved but not syntax checked.

2 A macro call forces the assembler to invoke the macro processor (expander). The macro expander switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander. The macro expander takes its input from the requested macro definition.

   The macro expander has no knowledge of assembler symbols since it only deals with text substitutions at source level. Before a line from the called macro definition is handed over to the assembler, the expander scans the line for all occurrences of symbolic macro arguments, and replaces them with their expansion arguments.

3 The expanded line is then processed as any other assembler source line. The input stream to the assembler continues to be the output from the macro processor, until all lines of the current macro definition have been read.

Repeating statements

Use the `REPT...ENDR` structure to assemble the same block of instructions several times. If *`expr`* evaluates to 0 nothing is generated.

Use `REPTC` to assemble a block of instructions once for each character in a string. If the string contains a comma it should be enclosed in quotation marks.

Only double quotes have a special meaning and their only use is to enclose the characters to iterate over. Single quotes have no special meaning and are treated as any ordinary character.

Use `REPTI` to assemble a block of instructions once for each string in a series of strings. Strings containing commas should be enclosed in quotation marks.

EXAMPLES

This section gives examples of the different ways in which macros can make assembler programming easier.

Coding inline for efficiency

In time-critical code it is often desirable to code routines inline to avoid the overhead of a subroutine call and return. Macros provide a convenient way of doing this.

This example outputs bytes from a buffer to a port:

```
            name    ioBufferSubroutine
            public  copyBuffer
ptbd        equ     0x0002          ; Definition of the port B
                                    ; data register.
            section DATA16:DATA
buffer      ds8     256

            section __DEFAULT_CODE_SECTION__:CODE(2)
copyBuffer  movi20  #256,r0          ; Initialize the loop
                                     ; counter.
            mov.l   #buffer,r1
            mov.l   #ptbd,r2
loop
            mov.b   @r1+,r3
            mov.b   r3,@r2
            cmp/pz  r0
            bt     loop             ; Have we copied 256 bytes?
            rts/n
            pool
            end
```

The main program calls this routine as follows:

```
            name    outputChars
            extern  copyBuffer
            public  main

            section __DEFAULT_CODE_SECTION__:CODE(2)
main        mov.l   #copyBuffer,r0
            jsr     @r0
            nop
            end
```

For efficiency we can recode this using a macro:

```
            name    ioBufferInline
ptbd        equ     0x0002          ; Definition of the port B
                                    ; data register.
            section DATA16:DATA
buffer      ds8     256

copyBuffer  macro
            local   loop
            movi20  #256,r0          ; Initialize the loop
                                     ; counter.
            mov.l   #buffer,r1
            mov.l   #ptbd,r2
```

```
loop
            mov.b   @r1+,r3
            mov.b   r3,@r2
            cmp/pz  r0
            bt      loop            ; Have we copied 256 bytes?
            endm


            section    __DEFAULT_CODE_SECTION__:CODE(2)
            copyBuffer
            rts/n
            end
```

Notice the use of the `LOCAL` directive to make the label `loop` local to the macro; otherwise an error is generated if the macro is used twice, as the `loop` label already exists.

Using REPTC and REPTI

This example assembles a series of calls to a subroutine `plot` to plot each character in a string:

```
            name    reptc
            extern  plotc
            section __DEFAULT_CODE_SECTION__:CODE(2)

banner      mov.l   #plotc,r1
            reptc   chr, "Welcome"
            mov     #'chr',r0
            jsr     @r1
            endr
            nop
            rts/n
            end
```

This produces this code:

```
     9                                  name    reptc
    10    000000                        extern  plotc
    11    000000                        section
                                        __DEFAULT_CODE_SECTION__:CODE(2)
    12
    13    000000 D108         banner     mov.l   #plotc,r1
    14    000002                         reptc   chr, "Welcome"
  14.1    000002 E057                    mov     #'W',r0
  14.2    000004 410B                    jsr     @r1
  14.3    000006 E065                    mov     #'e',r0
  14.4    000008 410B                    jsr     @r1
  14.5    00000A E06C                    mov     #'l',r0
```

```
14.6  00000C 410B                  jsr     @r1
14.7  00000E E063                  mov     #'c',r0
14.8  000010 410B                  jsr     @r1
14.9  000012 E06F                  mov     #'o',r0
14.10 000014 410B                  jsr     @r1
14.11 000016 E06D                  mov     #'m',r0
14.12 000018 410B                  jsr     @r1
14.13 00001A E065                  mov     #'e',r0
14.14 00001C 410B                  jsr     @r1
14.15 00001E                       endr
18    00001E 0009                  nop
19    000020 006B                  rts/n
20    000022 0000          ALIGN 2
20    000024               LONG TABLE
20    000024 00000000      Reference on line 13

20    000028                       end
```

This example uses `REPTI` to clear several memory locations:

```
            name    repti
            extern  base, count, init
            section __DEFAULT_CODE_SECTION__:CODE(2)

banner      mov     #0,r1
            repti   adds, base, count, init
            mov.l   #adds,r0
            mov.l   r1,@r0
            endr
            rts/n
            end
```

This produces this code:

```
 9                                 name    repti
10    000000                      extern  base, count, init
11
12    000000                      section
                                  __DEFAULT_CODE_SECTION__:CODE(2)
13
14    000000 E100         banner  mov     #0,r1
15    000002                      repti   adds, base, count, init
15.1  000002 D003                 mov.l   #base,r0
15.2  000004 2012                 mov.l   r1,@r0
15.3  000006 D003                 mov.l   #count,r0
15.4  000008 2012                 mov.l   r1,@r0
15.5  00000A D003                 mov.l   #init,r0
15.6  00000C 2012                 mov.l   r1,@r0
15.7  00000E                      endr
```

```
19    00000E 006B                    rts/n
20    000010               ALIGN 2
20    000010               LONG TABLE
20    000010 00000000      Reference on line 16

20    000014 00000000      Reference on line 16

20    000018 00000000      Reference on line 16

21    000004                         end
```

Listing control directives

These directives provide control over the assembler list file.

Directive	Description
LSTCND	Controls conditional assembly listing.
LSTCOD	Controls multi-line code listing.
LSTEXP	Controls the listing of macro-generated lines.
LSTMAC	Controls the listing of macro definitions.
LSTOUT	Controls assembly-listing output.
LSTREP	Controls the listing of lines generated by repeat directives.
LSTXRF	Generates a cross-reference table.

Table 23: Listing control directives

Note: The directives COL, LSTPAGE, PAGE, and PAGSIZ are included for backward compatibility reasons; they are recognized but no action is taken.

SYNTAX

```
LSTCND{+|-}
LSTCOD{+|-}
LSTEXP{+|-}
LSTMAC{+|-}
LSTOUT{+|-}
LSTREP{+|-}
LSTXRF{+|-}
```

DESCRIPTIONS

Turning the listing on or off

Use LSTOUT- to disable all list output except error messages. This directive overrides all other listing control directives.

The default is LSTOUT+, which lists the output (if a list file was specified).

Listing conditional code and strings

Use LSTCND+ to force the assembler to list source code only for the parts of the assembly that are not disabled by previous conditional IF statements.

The default setting is LSTCND-, which lists all source lines.

Use LSTCOD+ to list more than one line of code for a source line, if needed; that is, long ASCII strings produce several lines of output.

The default setting is LSTCOD-, which restricts the listing of output code to just the first line of code for a source line.

Using the LSTCND and LSTCOD directives does *not* affect code generation.

Controlling the listing of macros

Use LSTEXP- to disable the listing of macro-generated lines. The default is LSTEXP+, which lists all macro-generated lines.

Use LSTMAC+ to list macro definitions. The default is LSTMAC-, which disables the listing of macro definitions.

Controlling the listing of generated lines

Use LSTREP- to turn off the listing of lines generated by the directives REPT, REPTC, and REPTI.

The default is LSTREP+, which lists the generated lines.

Generating a cross-reference table

Use LSTXRF+ to generate a cross-reference table at the end of the assembler list for the current module. The table shows values and line numbers, and the type of the symbol.

The default is LSTXRF-, which does not give a cross-reference table.

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

```
            lstout-
            ; This section has already been debugged.
            lstout+
            ; This section is currently being debugged.
            end
```

Listing conditional code and strings

This example shows how `LSTCND+` hides a call to a subroutine that is disabled by an `IF` directive:

```
            name    lstcndTest
            extern  print
            section FLASH:CODE

debug       set     0
begin       if      debug
            bsr     print
            endif

            lstcnd+
begin2      if      debug
            bsr     print
            endif

            end
```

This generates the following listing:

```
 9                                      name    lstcndTest
10    000000                           extern  print
11    000000                           section FLASH:CODE
12
13    000000             debug         set     0
14    000000             begin         if      debug
15                                     bsr     print
16    000000                           endif
17
18                                     lstcnd+
19    000000             begin2        if      debug
21    000000                           endif
22
23    000000                           end
```

Controlling the listing of macros

This example shows the effect of `LSTMAC` and `LSTEXP`:

```
            name    lstmacTest
            extern  memLoc
            section FLASH:CODE(2)

dec2        macro   arg
            mov.l   #arg,r0
            mov.l   @r0,r1
            add     #-2,r1
            mov.l   r1,@r0
            endm

            lstmac+
inc2        macro   arg
            mov.l   #arg,r0
            mov.l   @r0,r1
            add     #2,r1
            mov.l   r1,@r0
            endm

begin       dec2    memLoc
            lstexp-
            inc2    memLoc
            rts/n
; Restore default values for
; listing control directives.

            lstmac-
            lstexp+

            end
```

This produces the following output:

```
      9                                     name    lstmacTest
     10    000000                           extern  memLoc
     11    000000                           section FLASH:CODE(2)
     12
     19
     20                                     lstmac+
     21                         inc2        macro   arg
     22                                     mov.l   #arg,r0
     23                                     mov.l   @r0,r1
     24                                     add     #2,r1
     25                                     mov.l   r1,@r0
     26                                     endm
```

```
27
28    000000              begin       dec2    memLoc
28.1  000000 D004                     mov.l   #memLoc,r0
28.2  000002 6102                     mov.l   @r0,r1
28.3  000004 71FE                     add     #-2,r1
28.4  000006 2012                     mov.l   r1,@r0
29                                    lstexp-
30    000008                          inc2    memLoc
31    000010 006B                     rts/n
32                        ; Restore default values for
33                        ; listing control directives.
34
35                                    lstmac-
36                                    lstexp+
37
38    000012 0000         ALIGN 2
38    000014              LONG TABLE
38    000014 00000000     Reference on line 14

38    000018 00000000     Reference on line 22

38    00001C                          end
```

C-style preprocessor directives

The assembler has a C-style preprocessor that follows the C99 standard.

These C-language preprocessor directives are available:

Directive	Description
`#define`	Assigns a value to a preprocessor symbol.
`#elif`	Introduces a new condition in an `#if...#endif` block.
`#else`	Assembles instructions if a condition is false.
`#endif`	Ends an `#if`, `#ifdef`, or `#ifndef` block.
`#error`	Generates an error.
`#if`	Assembles instructions if a condition is true.
`#ifdef`	Assembles instructions if a preprocessor symbol is defined.
`#ifndef`	Assembles instructions if a preprocessor symbol is undefined.
`#include`	Includes a file.
`#line`	Changes the source references in the debug information.

Table 24: C-style preprocessor directives

Directive	Description
`#pragma`	Controls extension features. The supported `#pragma` directives are described in the chapter *Pragma directives*.
`#undef`	Undefines a preprocessor symbol.

Table 24: C-style preprocessor directives (Continued)

SYNTAX

```
#define symbol text
#elif condition
#else
#endif
#error "message"
#if condition
#ifdef symbol
#ifndef symbol
#include {"filename" | <filename>}
#line line-no {"filename"}
#undef symbol
```

PARAMETERS

condition	An absolute expression	The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true.
filename	Name of file to be included or referred.	
line-no	Source line number.	
message	Text to be displayed.	
symbol	Preprocessor symbol to be defined, undefined, or tested.	
text	Value to be assigned.	

DESCRIPTIONS

You must not mix assembler language and C-style preprocessor directives. Conceptually, they are different languages and mixing them might lead to unexpected

behavior because an assembler directive is not necessarily accepted as a part of the C preprocessor language.

Note that the preprocessor directives are processed before other directives. As an example avoid constructs like:

```
redef       macro                   ; Avoid the following!
#define \1 \2
            endm
```

because the `\1` and `\2` macro arguments are not available during the preprocessing phase.

Defining and undefining preprocessor symbols

Use `#define` to define a value of a preprocessor symbol.

`#define` *`symbol value`*

Use `#undef` to undefine a symbol; the effect is as if it had not been defined.

Conditional preprocessor directives

Use the `#if...#else...#endif` directives to control the assembly process at assembly time. If the condition following the `#if` directive is not true, the subsequent instructions will not generate any code (that is, it will not be assembled or syntax checked) until an `#endif` or `#else` directive is found.

All assembler directives (except for `END`) and file inclusion can be disabled by the conditional directives. Each `#if` directive must be terminated by an `#endif` directive. The `#else` directive is optional and, if used, it must be inside an `#if...#endif` block.

`#if...#endif` and `#if...#else...#endif` blocks can be nested to any level.

Use `#ifdef` to assemble instructions up to the next `#else` or `#endif` directive only if a symbol is defined.

Use `#ifndef` to assemble instructions up to the next `#else` or `#endif` directive only if a symbol is undefined.

Including source files

Use `#include` to insert the contents of a file into the source file at a specified point. The filename can be specified within double quotes or within angle brackets.

Following is the full description of the assembler's `#include` file search procedure:

- If the name of the `#include` file is an absolute path, that file is opened.

- When the assembler encounters the name of an `#include` file in angle brackets such as:

  ```
  #include <io7201.h>
  ```

  it searches the following directories for the file to include:

  1 The directories specified with the `-I` option, in the order that they were specified.

  2 The directories specified using the `IASMSH_INC` environment variable, if any.

- When the assembler encounters the name of an `#include` file in double quotes such as:

  ```
  #include "vars.h"
  ```

  it searches the directory of the source file in which the `#include` statement occurs, and then performs the same sequence as for angle-bracketed filenames.

  If there are nested `#include` files, the assembler starts searching the directory of the file that was last included, iterating upwards for each included file, searching the source file directory last.

Use angle brackets for header files provided with the assembler, and double quotes for header files that are part of your application.

Displaying errors

Use `#error` to force the assembler to generate an error, such as in a user-defined test.

Comments in C-style preprocessor directives

If you make a comment within a define statement, use:

- the C comment delimiters `/* ... */` to comment sections
- the C++ comment delimiter `//` to mark the rest of the line as comment.

Do not use assembler comments within a define statement as it leads to unexpected behavior.

This expression evaluates to 3 because the comment character is preserved by `#define`:

```
#define x 3    ; This is a misplaced comment.

            module  misplacedComment1
expression  equ     x * 8 + 5
            ;...
            end
```

This example illustrates some problems that might occur when assembler comments are used in the C-style preprocessor:

```
#define five  5    ; This comment is not OK.
#define six   6    // This comment is OK.
#define seven 7    /* This comment is OK. */

            DC32    five, 11, 12
; The previous line expands to:
;           "DC32    5    ; This comment is not OK., 11, 12"

            DC32    six + seven, 11, 12
; The previous line expands to:
;           "DC32    6 + 7, 11, 12"

            end
```

Changing the source line numbers

Use the `#line` directive to change the source line numbers and the source filename used in the debug information. `#line` operates on the lines following the `#line` directive.

EXAMPLES

Using conditional preprocessor directives

This example defines the labels `tweak` and `adjust`. If `adjust` is defined, then register `16` is decremented by an amount that depends on `adjust`, in this case 30.

```
            module  calibrate
            extern  calibrationConstant
            section __DEFAULT_CODE_SECTION__:CODE(2)

#define     tweak  1
#define     adjust 3

calibrate   mov.l   #calibrationConstant,r0
            mov.b   @r0,r1
#ifdef      tweak
#if         adjust==1
            add     #-4,r1
#elif       adjust==2
            add     #-20,r1
#elif       adjust==3
            add     #-30,r1
#endif
#endif                                  /* ifdef tweak */
            mov.b   r1,@r0
```

```
            rts/n

            end
```

Including a source file

This example uses `#include` to include a file defining macros into the source file. For example, these macros could be defined in `Macros.inc`:

```
; Exchange registers a and b.

xch         macro   a,b
            xor \1,\2
            xor \2,\1
            xor \1,\2
            endm
```

The macro definitions can then be included, using `#include`, as in this example:

```
            program includeFile
            public xchRegs
            section __DEFAULT_CODE_SECTION__:CODE(2)

; Standard macro definitions
#include "Macros.inc"

xchRegs     xch     r1,r3
            xch     r2,r4
            rts/n
            end
```

Data definition or allocation directives

These directives define values or reserve memory. The column *Alias* in the following table shows the Renesas directive that corresponds to the IAR Systems directive. See *Expression restrictions*, page 13, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description
DC8	Generates 8-bit constants, including strings.
DC16	Generates 16-bit constants.
DC24	Generates 24-bit constants.
DC32	Generates 32-bit constants.
DC64	Generates 64-bit constants.

Table 25: Data definition or allocation directives

Directive	Description
DF32	Generates 32-bit floating-point constants.
DF64	Generates 64-bit floating-point constants.
DQ15	Generates 16-bit fractional constants.
DQ31	Generates 32-bit fractional constants.
DS8	Allocates space for 8-bit integers.
DS16	Allocates space for 16-bit integers.
DS24	Allocates space for 24-bit integers.
DS32	Allocates space for 32-bit integers.
DS64	Allocates space for 64-bit integers.

Table 25: Data definition or allocation directives (Continued)

SYNTAX

```
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DC64 expr [,expr] ...
DF32 value [,value] …
DF64 value [,value] …
DQ15 value [,value] …
DQ31 value [,value] …
DS8 count
DS16 count
DS24 count
DS32 count
DS64 count
```

PARAMETERS

count	A valid absolute expression specifying the number of elements to be reserved.
expr	A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings are zero filled to a multiple of the data size implied by the directive. Double-quoted strings are zero-terminated.*
value	A valid absolute expression or floating-point constant.

* **For** `DC64`**, the** `expr` **cannot be relocatable or external.**

DESCRIPTIONS

Use `DC8`, `DC16`, `DC24`, `DC32`, `DC64`, `DF32`, or `DF64` to create a constant, which means an area of bytes is reserved big enough for the constant.

Use `DS8`, `DS16`, `DS24`, `DS32`, or `DS64`, to reserve a number of uninitialized bytes.

EXAMPLES

Generating a lookup table

This example generates a constant table of 8-bit data that is accessed via the `call` instruction and added up to a sum.

```
            module  sumTableAndIndex
            section DATA16_C:CONST

table       dc8     12
            dc8     15
            dc8     17
            dc8     16
            dc8     14
            dc8     11
            dc8     9

            section __DEFAULT_CODE_SECTION__:CODE(2)
count       set     0

addTable    mov     #0,r2
            mov.l   #table,r1

            rept    7
            if      count == 7
            exitm
            endif
```

```
            mov.b   @(count,r1),r0
            add     r2,r0
count       set     count + 1
            endr

            rts/n

            end
```

Defining strings

To define a string:

```
myMsg   DC8 'Please enter your name'
```

To define a string which includes a trailing zero:

```
myCstr  DC8 "This is a string."
```

To include a single quote in a string, enter it twice; for example:

```
errMsg  DC8 'Don''t understand!'
```

Reserving space

To reserve space for `10` bytes:

```
table   DS8   10
```

Assembler control directives

These directives provide control over the operation of the assembler. See *Expression restrictions*, page 13, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
`/*comment*/`	C-style comment delimiter.	
`//`	C++ style comment delimiter.	
`CASEOFF`	Disables case sensitivity.	
`CASEON`	Enables case sensitivity.	
`POOL`	Specifies where to place constant tables.	
`RADIX`	Sets the default base on all numeric values.	No forward references No external references Absolute Fixed

Table 26: Assembler control directives

SYNTAX

```
/*comment*/
//comment
CASEOFF
CASEON
POOL
RADIX expr
```

PARAMETERS

comment	Comment ignored by the assembler.
expr	Default base; default 10 (decimal).

DESCRIPTIONS

Use `/*...*/` to comment sections of the assembler listing.

Use `//` to mark the rest of the line as comment.

Use `POOL` to make it possible to specify where to store constant tables.

Use `RADIX` to set the default base for constants. The default base is 10.

Controlling case sensitivity

Use `CASEON` or `CASEOFF` to turn on or off case sensitivity for user-defined symbols. By default, case sensitivity is on.

When `CASEOFF` is active all symbols are stored in upper case, and all symbols used by ILINK should be written in upper case in the ILINK definition file.

EXAMPLES

Defining comments

This example shows how `/*...*/` can be used for a multi-line comment:

```
/*
Program to read serial input.
Version 1: 19.2.02
Author: mjp
*/
```

See also, *Comments in C-style preprocessor directives*, page 84.

Specifying the location for constant tables

To specify the location for constant tables:

```
            module  caseSensitivity1
            public  initReg
            section __DEFAULT_CODE_SECTION__:CODE(2)

initReg     mov.w   #1234,R0
            mov.l   #12345678,R1
            pool                    ; Create constant tables for
                                    ; 1234 and 12345678
            mov.l   #9876543,R2
            rts/n
            end                     ; A constant table for 9876543
                                    ; will be created before end
```

For example, these instructions:

```
            mov.w   #1234,R0
            mov.l   #124345,R1
```

will expand to:

```
            mov.w   @(disp,PC),R0
            mov.l   @(disp,PC),R1

            dc16   1234
            dc32   124345
```

Changing the base

To set the default base to 16:

```
            module  radix
            section __DEFAULT_CODE_SECTION__:CODE(2)

            radix   16              ; With the default base set
            mov     #12,R1          ; to 16, the immediate value
            ;...                    ; of the load instruction is
                                    ; interpreted as 0x12.

; To reset the base from 16 to 10 again, the argument must be
; written in hexadecimal format.

            radix   0x0a            ; Reset the default base to 10
            mov     #12,R2          ; Now, the immediate value of
            ;...                    ; the load instruction is
                                    ; interpreted as 0x0c.
            end
```

Controlling case sensitivity

When `CASEOFF` is set, `label` and `LABEL` are identical in this example:

```
            module  caseSensitivity1
            section __DEFAULT_CODE_SECTION__:CODE(2)

            caseoff
label       nop                     ; Stored as "LABEL".
            bra     LABEL
            nop
            end
```

The following will generate a duplicate label error:

```
            module  caseSensitivity2
            section __DEFAULT_CODE_SECTION__:CODE(2)

            caseoff
label       nop                     ; Stored as "LABEL".
LABEL       nop                     ; Error, "LABEL" already
                                    ; defined.
            end
```

Call frame information directives

These directives allow backtrace information to be defined in the assembler source code. The benefit is that you can view the call frame stack when you debug your assembler code.

Directive	Description
`CFI BASEADDRESS`	Declares a base address CFA (Canonical Frame Address).
`CFI BLOCK`	Starts a data block.
`CFI CODEALIGN`	Declares code alignment.
`CFI COMMON`	Starts or extends a common block.
`CFI CONDITIONAL`	Declares data block to be a conditional thread.
`CFI DATAALIGN`	Declares data alignment.
`CFI ENDBLOCK`	Ends a data block.
`CFI ENDCOMMON`	Ends a common block.
`CFI ENDNAMES`	Ends a names block.
`CFI FRAMECELL`	Creates a reference into the caller's frame.
`CFI FUNCTION`	Declares a function associated with data block.

Table 27: Call frame information directives

Directive	Description
CFI INVALID	Starts range of invalid backtrace information.
CFI NAMES	Starts a names block.
CFI NOFUNCTION	Declares data block to not be associated with a function.
CFI PICKER	Declares data block to be a picker thread.
CFI REMEMBERSTATE	Remembers the backtrace information state.
CFI RESOURCE	Declares a resource.
CFI RESOURCEPARTS	Declares a composite resource.
CFI RESTORESTATE	Restores the saved backtrace information state.
CFI RETURNADDRESS	Declares a return address column.
CFI STACKFRAME	Declares a stack frame CFA.
CFI STATICOVERLAYFRAME	Declares a static overlay frame CFA.
CFI VALID	Ends range of invalid backtrace information.
CFI VIRTUALRESOURCE	Declares a virtual resource.
CFI *cfa*	Declares the value of a CFA.
CFI *resource*	Declares the value of a resource.

Table 27: Call frame information directives (Continued)

SYNTAX

The syntax definitions below show the syntax of each directive. The directives are grouped according to usage.

Names block directives

```
CFI NAMES name
CFI ENDNAMES name
CFI RESOURCE resource : bits [, resource : bits] …
CFI VIRTUALRESOURCE resource : bits [, resource : bits] …
CFI RESOURCEPARTS resource part, part [, part] ...
CFI STACKFRAME cfa resource type [, cfa resource type] …
CFI STATICOVERLAYFRAME cfa section [, cfa section] …
CFI BASEADDRESS cfa type [, cfa type] …
```

Extended names block directives

```
CFI NAMES name EXTENDS namesblock
CFI ENDNAMES name
CFI FRAMECELL cell cfa (offset): size [, cell cfa (offset) : size] …
```

Common block directives

```
CFI COMMON name USING namesblock
CFI ENDCOMMON name
CFI CODEALIGN codealignfactor
CFI DATAALIGN dataalignfactor
CFI RETURNADDRESS resource type
CFI cfa { NOTUSED | USED }
CFI cfa { resource | resource + constant | resource - constant }
CFI cfa cfiexpr
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME(cfa, offset) }
CFI resource cfiexpr
```

Extended common block directives

```
CFI COMMON name EXTENDS commonblock USING namesblock
CFI ENDCOMMON name
```

Data block directives

```
CFI BLOCK name USING commonblock
CFI ENDBLOCK name
CFI { NOFUNCTION | FUNCTION label }
CFI { INVALID | VALID }
CFI { REMEMBERSTATE | RESTORESTATE }
CFI PICKER
CFI CONDITIONAL label [, label] …
CFI cfa { resource | resource + constant | resource - constant }
CFI cfa cfiexpr
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME(cfa, offset) }
CFI resource cfiexpr
```

PARAMETERS

bits	The size of the resource in bits.
cell	The name of a frame cell.
cfa	The name of a CFA (canonical frame address).

cfiexpr	A CFI expression (see *CFI expressions*, page 101).
codealignfactor	The smallest factor of all instruction sizes. Each CFI directive for a data block must be placed according to this alignment. 1 is the default and can always be used, but a larger value shrinks the produced backtrace information in size. The possible range is 1–256.
commonblock	The name of a previously defined common block.
constant	A constant value or an assembler expression that can be evaluated to a constant value.
dataalignfactor	The smallest factor of all frame sizes. If the stack grows toward higher addresses, the factor is negative; if it grows toward lower addresses, the factor is positive. 1 is the default, but a larger value shrinks the produced backtrace information in size. The possible ranges are -256 to -1 and 1 to 256.
label	A function label.
name	The name of the block.
namesblock	The name of a previously defined names block.
offset	The offset relative the CFA. An integer with an optional sign.
part	A part of a composite resource. The name of a previously declared resource.
resource	The name of a resource.
section	The name of a section.
size	The size of the frame cell in bytes.
type	The memory type, such as `CODE`, `CONST` or `DATA`. In addition, any of the memory types supported by the IAR ILINK Linker. It is used solely for the purpose of denoting an address space.

DESCRIPTIONS

The call frame information directives (CFI directives) are an extension to the debugging format of the IAR C-SPY® Debugger. The CFI directives are used for defining the *backtrace information* for the instructions in a program. The compiler normally generates this information, but for library functions and other code written purely in assembler language, backtrace information must be added if you want to use the call frame stack in the debugger.

The backtrace information is used to keep track of the contents of *resources*, such as registers or memory cells, in the assembler code. This information is used by the IAR C-SPY Debugger to go "back" in the call stack and show the correct values of registers or other resources before entering the function. In contrast with traditional approaches, this permits the debugger to run at full speed until it reaches a breakpoint, stop at the breakpoint, and retrieve backtrace information at that point in the program. The information can then be used to compute the contents of the resources in any of the calling functions—assuming they have call frame information as well.

Backtrace rows and columns

At each location in the program where it is possible for the debugger to break execution, there is a *backtrace row*. Each backtrace row consists of a set of *columns*, where each column represents an item that should be tracked. There are three kinds of columns:

- The *resource columns* keep track of where the original value of a resource can be found.
- The canonical frame address columns (*CFA columns*) keep track of the top of the function frames.
- The *return address column* keeps track of the location of the return address.

There is always exactly one return address column and usually only one CFA column, although there might be more than one.

Defining a names block

A *names block* is used to declare the resources available for a processor. Inside the names block, all resources that can be tracked are defined.

Start and end a names block with the directives:

```
CFI NAMES name
CFI ENDNAMES name
```

where *name* is the name of the block.

Only one names block can be open at a time.

Inside a names block, four different kinds of declarations can appear: a resource declaration, a stack frame declaration, a static overlay frame declaration, or a base address declaration:

- To declare a resource, use one of the directives:

  ```
  CFI RESOURCE resource : bits
  CFI VIRTUALRESOURCE resource : bits
  ```

The parameters are the name of the resource and the size of the resource in bits. A virtual resource is a logical concept, in contrast to a "physical" resource such as a processor register. Virtual resources are usually used for the return address.

To declare more than one resource, separate them with commas.

A resource can also be a composite resource, made up of at least two parts. To declare the composition of a composite resource, use the directive:

```
CFI RESOURCEPARTS resource part, part, …
```

The parts are separated with commas. The resource and its parts must have been previously declared as resources, as described above.

- To declare a stack frame CFA, use the directive:

  ```
  CFI STACKFRAME cfa resource type
  ```

  The parameters are the name of the stack frame CFA, the name of the associated resource (the stack pointer), and the section type (to get the address space). To declare more than one stack frame CFA, separate them with commas.

  When going "back" in the call stack, the value of the stack frame CFA is copied into the associated stack pointer resource to get a correct value for the previous function frame.

- To declare a static overlay frame CFA, use the directive:

  ```
  CFI STATICOVERLAYFRAME cfa section
  ```

  The parameters are the name of the CFA and the name of the section where the static overlay for the function is located. To declare more than one static overlay frame CFA, separate them with commas.

- To declare a base address CFA, use the directive:

  ```
  CFI BASEADDRESS cfa type
  ```

  The parameters are the name of the CFA and the section type. To declare more than one base address CFA, separate them with commas.

  A base address CFA is used to conveniently handle a CFA. In contrast to the stack frame CFA, there is no associated stack pointer resource to restore.

Extending a names block

In some special cases you must extend an existing names block with new resources. This occurs whenever there are routines that manipulate call frames other than their own, such as routines for handling, entering, and leaving C or C++ functions; these routines manipulate the caller's frame. Extended names blocks are normally used only by compiler developers.

Extend an existing names block with the directive:

```
CFI NAMES name EXTENDS namesblock
```

where *namesblock* is the name of the existing names block and *name* is the name of the new extended block. The extended block must end with the directive:

```
CFI ENDNAMES name
```

Defining a common block

The *common block* is used for declaring the initial contents of all tracked resources. Normally, there is one common block for each calling convention used.

Start a common block with the directive:

```
CFI COMMON name USING namesblock
```

where *name* is the name of the new block and *namesblock* is the name of a previously defined names block.

Declare the return address column with the directive:

```
CFI RETURNADDRESS resource type
```

where *resource* is a resource defined in *namesblock* and *type* is the section type. You must declare the return address column for the common block.

End a common block with the directive:

```
CFI ENDCOMMON name
```

where *name* is the name used to start the common block.

Inside a common block, you can declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 94. For more information on these directives, see *Simple rules*, page 99, and *CFI expressions*, page 101.

Extending a common block

Since you can extend a names block with new resources, it is necessary to have a mechanism for describing the initial values of these new resources. For this reason, it is also possible to extend common blocks, effectively declaring the initial values of the extra resources while including the declarations of another common block. Just as in the case of extended names blocks, extended common blocks are normally only used by compiler developers.

Extend an existing common block with the directive:

```
CFI COMMON name EXTENDS commonblock USING namesblock
```

where *name* is the name of the new extended block, *commonblock* is the name of the existing common block, and *namesblock* is the name of a previously defined names block. The extended block must end with the directive:

```
CFI ENDCOMMON name
```

Defining a data block

The *data block* contains the actual tracking information for one continuous piece of code. No section control directive can appear inside a data block.

Start a data block with the directive:

```
CFI BLOCK name USING commonblock
```

where *name* is the name of the new block and *commonblock* is the name of a previously defined common block.

If the piece of code is part of a defined function, specify the name of the function with the directive:

```
CFI FUNCTION label
```

where *label* is the code label starting the function.

If the piece of code is not part of a function, specify this with the directive:

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

where *name* is the name used to start the data block.

Inside a data block, you can manipulate the values of the columns by using the directives listed last in *Data block directives*, page 94. For more information on these directives, see *Simple rules*, page 99, and *CFI expressions*, page 101.

SIMPLE RULES

To describe the tracking information for individual columns, there is a set of simple rules with specialized syntax:

```
CFI cfa { NOTUSED | USED }
CFI cfa { resource | resource + constant | resource - constant }
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
CFI resource { resource | FRAME(cfa, offset) }
```

You can use these simple rules both in common blocks to describe the initial information for resources and CFAs, and inside data blocks to describe changes to the information for resources or CFAs.

In those rare cases where the descriptive power of the simple rules are not enough, you can use a full CFI expression to describe the information (see *CFI expressions*, page 101). However, whenever possible, you should always use a simple rule instead of a CFI expression.

There are two different sets of simple rules: one for resources and one for CFAs.

Simple rules for resources

The rules for resources conceptually describe where to find a resource when going back one call frame. For this reason, the item following the resource name in a CFI directive is referred to as the *location* of the resource.

To declare that a tracked resource is restored, that is, already correctly located, use `SAMEVALUE` as the location. Conceptually, this declares that the resource does not have to be restored since it already contains the correct value. For example, to declare that a register `REG` is restored to the same value, use the directive:

```
CFI REG SAMEVALUE
```

To declare that a resource is not tracked, use `UNDEFINED` as location. Conceptually, this declares that the resource does not have to be restored (when going back one call frame) since it is not tracked. Usually it is only meaningful to use it to declare the initial location of a resource. For example, to declare that `REG` is a scratch register and does not have to be restored, use the directive:

```
CFI REG UNDEFINED
```

To declare that a resource is temporarily stored in another resource, use the resource name as its location. For example, to declare that a register `REG1` is temporarily located in a register `REG2` (and should be restored from that register), use the directive:

```
CFI REG1 REG2
```

To declare that a resource is currently located somewhere on the stack, use `FRAME(`*`cfa`*`, `*`offset`*`)` as location for the resource, where *`cfa`* is the CFA identifier to use as "frame pointer" and *`offset`* is an offset relative the CFA. For example, to declare that a register `REG` is located at offset -4 counting from the frame pointer `CFA_SP`, use the directive:

```
CFI REG FRAME(CFA_SP,-4)
```

For a composite resource there is one additional location, `CONCAT`, which declares that the location of the resource can be found by concatenating the resource parts for the composite resource. For example, consider a composite resource `RET` with resource parts `RETLO` and `RETHI`. To declare that the value of `RET` can be found by investigating and concatenating the resource parts, use the directive:

```
CFI RET CONCAT
```

This requires that at least one of the resource parts has a definition, using the rules described above.

Simple rules for CFAs

In contrast with the rules for resources, the rules for CFAs describe the address of the beginning of the call frame. The call frame often includes the return address pushed by the subroutine calling instruction. The CFA rules describe how to compute the address to the beginning of the current call frame. There are two different forms of CFAs, stack frames and static overlay frames, each declared in the associated names block. See *Names block directives*, page 93.

Each stack frame CFA is associated with a resource, such as the stack pointer. When going back one call frame the associated resource is restored to the current CFA. For stack frame CFAs there are two possible simple rules: an offset from a resource (not necessarily the resource associated with the stack frame CFA) or `NOTUSED`.

To declare that a CFA is not used, and that the associated resource should be tracked as a normal resource, use `NOTUSED` as the address of the CFA. For example, to declare that the CFA with the name `CFA_SP` is not used in this code block, use the directive:

```
CFI CFA_SP NOTUSED
```

To declare that a CFA has an address that is offset relative the value of a resource, specify the resource and the offset. For example, to declare that the CFA with the name `CFA_SP` can be obtained by adding 4 to the value of the `SP` resource, use the directive:

```
CFI CFA_SP SP + 4
```

For static overlay frame CFAs, there are only two possible declarations inside common and data blocks: `USED` and `NOTUSED`.

CFI EXPRESSIONS

You can use call frame information expressions (CFI expressions) when the descriptive power of the simple rules for resources and CFAs is not enough. However, you should always use a simple rule when one is available.

CFI expressions consist of operands and operators. Only the operators described below are allowed in a CFI expression. In most cases, they have an equivalent operator in the regular assembler expressions.

In the operand descriptions, *`cfiexpr`* denotes one of these:

- A CFI operator with operands
- A numeric constant
- A CFA name
- A resource name.

Unary operators

Overall syntax: *OPERATOR*(*operand*)

Operator	Operand	Description
COMPLEMENT	*cfiexpr*	Performs a bitwise NOT on a CFI expression.
LITERAL	*expr*	Get the value of the assembler expression. This can insert the value of a regular assembler expression into a CFI expression.
NOT	*cfiexpr*	Negates a logical CFI expression.
UMINUS	*cfiexpr*	Performs arithmetic negation on a CFI expression.

Table 28: Unary operators in CFI expressions

Binary operators

Overall syntax: *OPERATOR*(*operand1*,*operand2*)

Operator	Operands	Description
ADD	*cfiexpr*,*cfiexpr*	Addition
AND	*cfiexpr*,*cfiexpr*	Bitwise AND
DIV	*cfiexpr*,*cfiexpr*	Division
EQ	*cfiexpr*,*cfiexpr*	Equal
GE	*cfiexpr*,*cfiexpr*	Greater than or equal
GT	*cfiexpr*,*cfiexpr*	Greater than
LE	*cfiexpr*,*cfiexpr*	Less than or equal
LSHIFT	*cfiexpr*,*cfiexpr*	Logical shift left of the left operand. The number of bits to shift is specified by the right operand. The sign bit will not be preserved when shifting.
LT	*cfiexpr*,*cfiexpr*	Less than
MOD	*cfiexpr*,*cfiexpr*	Modulo
MUL	*cfiexpr*,*cfiexpr*	Multiplication
NE	*cfiexpr*,*cfiexpr*	Not equal
OR	*cfiexpr*,*cfiexpr*	Bitwise OR
RSHIFTA	*cfiexpr*,*cfiexpr*	Arithmetic shift right of the left operand. The number of bits to shift is specified by the right operand. In contrast with RSHIFTL, the sign bit is preserved when shifting.

Table 29: Binary operators in CFI expressions

Operator	Operands	Description
RSHIFTL	*cfiexpr*,*cfiexpr*	Logical shift right of the left operand. The number of bits to shift is specified by the right operand. The sign bit will not be preserved when shifting.
SUB	*cfiexpr*,*cfiexpr*	Subtraction
XOR	*cfiexpr*,*cfiexpr*	Bitwise XOR

Table 29: Binary operators in CFI expressions (Continued)

Ternary operators

Overall syntax: *OPERATOR*(*operand1*,*operand2*,*operand3*)

Operator	Operands	Description
FRAME	*cfa*,*size*,*offset*	Gets the value from a stack frame. The operands are: *cfa* An identifier denoting a previously declared CFA. *size* A constant expression denoting a size in bytes. *offset* A constant expression denoting an offset in bytes. Gets the value at address *cfa+offset* of size *size*.
IF	*cond*,*true*,*false*	Conditional operator. The operands are: *cond* A CFA expression denoting a condition. *true* Any CFA expression. *false* Any CFA expression. If the conditional expression is non-zero, the result is the value of the *true* expression; otherwise the result is the value of the *false* expression.
LOAD	*size*,*type*,*addr*	Gets the value from memory. The operands are: *size* A constant expression denoting a size in bytes. *type* A memory type. *addr* A CFA expression denoting a memory address. Gets the value at address *addr* in section type *type* of size *size*.

Table 30: Ternary operators in CFI expressions

EXAMPLE

The following is a generic example and not an example specific to the SH microprocessor. This simplifies the example and clarifies the usage of the CFI directives. To obtain a target-specific example, generate assembler output when you compile a C source file.

Consider a generic processor with a stack pointer SP, and two registers R0 and R1. Register R0 is used as a scratch register (the register is destroyed by the function call),

whereas register R1 must be restored after the function call. For reasons of simplicity, all instructions, registers, and addresses have a width of 16 bits.

Consider the following short code sample with the corresponding backtrace rows and columns. At entry, assume that the stack contains a 16-bit return address. The stack grows from high addresses toward zero. The CFA denotes the top of the call frame, that is, the value of the stack pointer after returning from the function.

Address	CFA	SP	R0	R1	RET	Assembler code
0000	SP + 2		—	SAME	CFA - 2	func1: PUSH R1
0002	SP + 4			CFA - 4		MOV R1,#4
0004						CALL func2
0006						POP R0
0008	SP + 2			R0		MOV R1,R0
000A				SAME		RET

Table 31: Code sample with backtrace rows and columns

Each backtrace row describes the state of the tracked resources *before* the execution of the instruction. As an example, for the MOV R1,R0 instruction the original value of the R1 register is located in the R0 register and the top of the function frame (the CFA column) is SP + 2. The backtrace row at address 0000 is the initial row and the result of the calling convention used for the function.

The SP column is empty since the CFA is defined in terms of the stack pointer. The RET column is the return address column—that is, the location of the return address. The R0 column has a '—' in the first line to indicate that the value of R0 is undefined and does not need to be restored on exit from the function. The R1 column has SAME in the initial row to indicate that the value of the R1 register will be restored to the same value it already has.

Defining the names block

The names block for the small example above would be:

```
CFI NAMES trivialNames
CFI RESOURCE SP:16, R0:16, R1:16
CFI STACKFRAME CFA SP DATA

;; The virtual resource for the return address column
CFI VIRTUALRESOURCE RET:16
CFI ENDNAMES trivialNames
```

Defining the common block

The common block for the simple example above would be:

```
CFI COMMON trivialCommon USING trivialNames
CFI RETURNADDRESS RET DATA
CFI CFA SP + 2
CFI R0 UNDEFINED
CFI R1 SAMEVALUE
CFI RET FRAME(CFA,-2)  ; Offset -2 from top of frame
CFI ENDCOMMON trivialCommon
```

Note: `SP` cannot be changed using a CFI directive since it is the resource associated with `CFA`.

Defining the data block

Continuing the simple example, the data block would be:

```
     RSEG   CODE:CODE
     CFI    BLOCK func1block USING trivialCommon
     CFI    FUNCTION func1
func1:
     PUSH   R1
     CFI    CFA SP + 4
     CFI    R1 FRAME(CFA,-4)
     MOV    R1,#4
     CALL   func2
     POP    R0
     CFI    R1 R0
     CFI    CFA SP + 2
     MOV    R1,R0
     CFI    R1 SAMEVALUE
     RET
     CFI ENDBLOCK func1block
```

Note that the CFI directives are placed *after* the instruction that affects the backtrace information.

Pragma directives

This chapter describes the pragma directives of the assembler.

The pragma directives control the behavior of the assembler, for example whether it outputs warning messages. The pragma directives are preprocessed, which means that macros are substituted in a pragma directive.

Summary of pragma directives

This table shows the pragma directives of the assembler:

#pragma directive	Description
`#pragma diag_default`	Changes the severity level of diagnostic messages
`#pragma diag_error`	Changes the severity level of diagnostic messages
`#pragma diag_remark`	Changes the severity level of diagnostic messages
`#pragma diag_suppress`	Suppresses diagnostic messages
`#pragma diag_warning`	Changes the severity level of diagnostic messages
`#pragma message`	Prints a message

Table 32: Pragma directives summary

Descriptions of pragma directives

All pragma directives using = for value assignment should be entered like:

```
#pragma pragmaname=pragmavalue
```

or

```
#pragma pragmaname = pragmavalue
```

#pragma diag_default

```
#pragma diag_default=tag,tag,...
```

Changes the severity level back to default or as defined on the command line for the diagnostic messages with the specified tags. For example:

```
#pragma diag_default=Pe117
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_error

```
#pragma diag_error=tag,tag,...
```

Changes the severity level to `error` for the specified diagnostics. For example:

```
#pragma diag_error=Pe117
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_remark

```
#pragma diag_remark=tag,tag,...
```

Changes the severity level to `remark` for the specified diagnostics. For example:

```
#pragma diag_remark=Pe177
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_suppress

```
#pragma diag_suppress=tag,tag,...
```

Suppresses the diagnostic messages with the specified tags. For example:

```
#pragma diag_suppress=Pe117,Pe177
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma diag_warning

```
#pragma diag_warning=tag,tag,...
```

Changes the severity level to `warning` for the specified diagnostics. For example:

```
#pragma diag_warning=Pe826
```

See the chapter *Diagnostics* for more information about diagnostic messages.

#pragma message

```
#pragma message(string)
```

Makes the assembler print a message on `stdout` when the file is assembled. For example:

```
#ifdef TESTING
#pragma message("Testing")
#endif
```

Diagnostics

This chapter describes the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

Message format

All diagnostic messages are issued as complete, self-explanatory messages. A typical diagnostic message from the assembler is produced in the form:

```
filename,linenumber level[tag]: message
```

where *filename* is the name of the source file in which the error was encountered; *linenumber* is the line number at which the assembler detected the error; *level* is the level of seriousness of the diagnostic; *tag* is a unique tag that identifies the diagnostic message; *message* is a self-explanatory message, possibly several lines long.

Diagnostic messages are displayed on the screen, and printed in the optional list file. In the IDE, diagnostic messages are displayed in the Build messages window.

Severity levels

The diagnostics are divided into different levels of severity:

Remark

A diagnostic message that is produced when the assembler finds a source code construct that can possibly lead to erroneous behavior in the generated code. Remarks are, by default, not issued but can be enabled, see *--remarks*, page 32.

Warning

A diagnostic message that is produced when the assembler finds a programming error or omission which is of concern but not so severe as to prevent the completion of compilation. Warnings can be disabled with the command line option `--no_warnings`, see *--no_warnings*, page 30.

Error

A diagnostic message that is produced when the assembler finds a construct which clearly violates the language rules, such that code cannot be produced. An error produces a non-zero exit code.

Fatal error

A diagnostic message that is produced when the assembler finds a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the diagnostic is issued, assembly ends. A fatal error produces a non-zero exit code.

SETTING THE SEVERITY LEVEL

The diagnostic messages can be suppressed or the severity level can be changed for all types of diagnostics except for fatal errors and some of the regular errors.

See *Summary of assembler options*, page 18, for a description of the assembler options that are available for setting severity levels.

See the chapter *Pragma directives*, for a description of the pragma directives that are available for setting severity levels.

INTERNAL ERROR

An internal error is a diagnostic message that signals that there was a serious and unexpected failure due to a fault in the assembler. It is produced using this form:

```
Internal error: message
```

where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR Systems Technical Support. Please include information enough to reproduce the problem. This would typically include:

- The product name
- The version number of the assembler, which can be seen in the header of the list files generated by the assembler
- Your license number
- The exact internal error message text
- The source file of the program that generated the internal error
- A list of the options that were used when the internal error occurred.

A

B

C

D

E

F

G

H

I

L

M

N

T

U

V

W

X

Symbols