IAR Embedded Workbench

IAR Assembler User Guide

for

RISC-V

ARISCV-2

IAR SYSTEMS

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS

IAR Systems, IAR Embedded Workbench, Embedded Trust, C-Trust, IAR Connect, C-SPY, C-RUN, C-STAT, IAR Visual State, IAR KickStart Kit, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

RISC-V is a trademark of the RISC-V Foundation, pending registration.

Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated.

All other product names are trademarks or registered trademarks of their respective owners.

EDITION NOTICE

Second edition: December 2019

Part number: ARISCV-2

This guide applies to version 1.x of IAR Embedded Workbench® for RISC-V.

Internal reference: BB6, asrct2010.3, V_110411, IJOA.

Contents

Tables

Preface

Welcome to the *IAR Assembler User Guide for RISC-V*. The purpose of this guide is to provide you with detailed reference information that can help you to use the IAR Assembler for RISC-V 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 RISC-V, and need to get detailed reference information on how to use the IAR Assembler for RISC-V. In addition, you should have working knowledge of the following:

- The architecture and instruction set of RISC-V (refer to the chip manufacturer's documentation or the RISC-V Foundation website—**riscv.org**)
- 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 RISC-V, you should read the chapter *Introduction to the IAR Assembler for RISC-V*.

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 Embedded Workbench, we recommend that you first work through the tutorials, which you can find in the IAR Information Center and which will help you get started using IAR Embedded Workbench.

What this guide contains

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

- *Introduction to the IAR Assembler for RISC-V* 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.

Document conventions

When, in the IAR Systems documentation, 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 `riscv\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench N.n\riscv\doc`, where the initial digit of the version number reflects the initial digit of the version number of the IAR Embedded Workbench shared components.

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set 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.

Table 1: Typographic conventions used in this guide

Style	Used for
[option**]**	An optional part of a linker or stack usage control directive, where **[** and **]** are not part of the actual directive, but any `[`, `]`, `{`, or `}` are part of the directive syntax.
{option**}**	A mandatory part of a linker or stack usage control directive, where **{** and **}** are not part of the actual directive, but any `[`, `]`, `{`, or `}` are part of the directive syntax.
`[option]`	An optional part of a command line option, pragma directive, or library filename.
`[a|b|c]`	An optional part of a command line option, pragma directive, or library filename with alternatives.
`{a|b|c}`	A mandatory part of a command line option, pragma directive, or library filename 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 (Continued)

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems®, when referred to in the documentation:

Brand name	Generic term
IAR Embedded Workbench® for RISC-V	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for RISC-V	the IDE
IAR C-SPY® Debugger for RISC-V	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator

Table 2: Naming conventions used in this guide

Brand name	Generic term
IAR C/C++ Compiler™ for RISC-V	the compiler
IAR Assembler™ for RISC-V	the assembler
IAR ILINK Linker™	ILINK, the linker
IAR DLIB Runtime Environment™	the DLIB runtime environment

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

Introduction to the IAR Assembler for RISC-V

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

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 RISC-V that require precise timing and special instruction sequences.

To write efficient assembler applications, you should be familiar with the architecture and instruction set of RISC-V. Refer to the documentation on the RISC-V Foundation website—**riscv.org**—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 Information Center
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the *IAR C/C++ Development Guide for RISC-V*

- 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 enable you to 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 IAR Embedded Workbench IDE, you can set up a library project, to collect many object files in one library. For an example, see the tutorials in the Information Center.

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:

- *Assembler invocation syntax*, page 17
- *Passing options*, page 18
- *Environment variables*, page 18
- *Error return codes*, page 18

You can use the assembler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *IDE Project Management and Building Guide for RISC-V* for information about using the assembler from the IAR Embedded Workbench IDE.

ASSEMBLER INVOCATION SYNTAX

The invocation syntax for the assembler is:

```
iasmriscv [options][sourcefile][options]
```

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

```
iasmriscv prog --debug
```

By default, the IAR Assembler for RISC-V 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 `iasmriscv` command, see *Assembler invocation syntax*, page 17.
- Via environment variables

  The assembler automatically appends the value of the environment variables to every command line, so it provides a convenient method of specifying options that are required for every assembly, see *Environment variables*, page 18.
- Via a text file by using the `-f` option, see *-f*, page 49.

For general guidelines for the option syntax, an options summary, and more information about each option, see the *Assembler options* chapter.

ENVIRONMENT VARIABLES

You can use these environment variables with the IAR Assembler:

Environment variable	Description
`IASMRISCV`	Specifies command line options, for example: `set IASMRISCV=la . --warnings_are_errors`
`IASMRISCV_INC`	Specifies directories to search for include files, for example: `set IASMRISCV_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 IASMRISCV=-l temp.lst
```

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

ERROR RETURN CODES

When using the IAR 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.

Table 4: Assembler error return codes

Return code	Description
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 (Continued)

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 or whitespaces. 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 memory operand on the form *x*(*reg*)
- 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 cannot exceed 2,047 characters.

Tab characters, ASCII `09H`, are expanded according to the most common practice, that is, 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 RISC-V supports the syntax for assembler instructions as described in the assembly documentation for the RISC-V ISA. It complies with the requirement of the RISC-V architecture on word alignment. Any instructions in a code section placed on an incorrectly aligned 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 64-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*.

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 details 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. If they are not resolved, the assembler generates an error.

INTEGER CONSTANTS

Because all IAR Systems assemblers use 64-bit two's complement internal arithmetic, integers have a (signed) range from -2^{63} to $2^{63}-1$.

Constants are written as a sequence of digits with an optional preceding - (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)	'\0' (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 accepts floating-point values as constants and converts them into IEEE single-precision (32-bit) floating-point format, double-precision (64-bit), or fractional format.

Floating-point numbers can be written in the format:

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

This table shows 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).

Symbols may contain any printable characters if they are quoted with ` (backquote), for example:

```
`strange#label`
```

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. For more information, see *--case_insensitive*, page 43.

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. See also *Data definition or allocation directives*, page 111.

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*.

To refer to the program location counter in your assembler source code, use the `$` (dollar) character. For example:

```
        j       $         ; Loop forever
```

REGISTER SYMBOLS

This table shows the existing predefined integer register symbols:

Name	Alias	Description
`x0`	`Zero`	Always zero
`x1`	`ra`	Return address
`x2`	`sp`	Stack pointer
`x3`	`gp`	Global pointer
`x4`	`tp`	Thread pointer
`x5`	`t0`	Temporary register/Alternate return address
`x6–x7`	`t1–t2`	Temporary register
`x8`	`s0/fp`	Saved register/Frame pointer
`x9`	`s1`	Saved register
`x10–x11`	`a0–a1`	Function argument/Return value
`x12–x17`	`a2–a7`	Function argument

Table 8: Predefined integer register symbols

Name	Alias	Description
`x18–x27`	`s2–s11`	Saved register
`x28–x31`	`t3–t6`	Temporary register

Table 8: Predefined integer register symbols (Continued)

This table shows the floating-point register symbols that are predefined for cores with the floating-point extension:

Name	Alias	Description
`f0–f7`	`ft0–ft7`	Floating-point temporaries
`f8–f9`	`fs0–fs1`	Floating-point saved registers
`f10`	`fa0`	Floating-point arguments/return values
`f11–f17`	`fa1–fa7`	Floating-point arguments
`f18–f27`	`fs2–fs11`	Floating-point saved registers
`f28–f31`	`ft3–ft6`	Floating-point temporaries

Table 9: Predefined floating-point register symbols

Note: The size of a floating-point register is equal to the precision of the FPU.

PREDEFINED SYMBOLS

The IAR Assembler for RISC-V 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.

These predefined symbols are available:

Symbol	Value
`__IASMRISCV__`	An integer that is set to `1` when the code is assembled with the IAR Assembler for RISC-V.
`__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.
`__DATE__`	The current date in `dd/Mmm/yyyy` format (string).
`__FILE__`	The name of the current source file (string).
`__IAR_SYSTEMS_ASM__`	IAR assembler identifier (number). The current value is 9. 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.

Table 10: Predefined symbols

Symbol	Value
__LINE__	The current source line number (number).
__riscv	An integer that is set to 1 when the code is assembled for RISC-V.
__riscv_32e	An integer that is set to 1 when the code is assembled for a RISC-V core with the E extension.
__riscv_compressed	An integer that is set to 1 when the code is assembled for a RISC-V core with the C extension.
__riscv_flen	An integer that is set to 32 when the code is assembled for a RISC-V core with the F (but not the D) extension, and to 64 when the code is assembled for a core with the FD extensions (implicitly or explicitly). If the code is assembled for neither extension, this symbol is undefined.
__riscv_xlen	An integer that is set to 32. This symbol identifies when the code is assembled for a 32-bit RISC-V core.
__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 10: 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
            rseg    `,text`:CODE(2)

printTime   lui     a0, %hi(time)        ; Load address of time
            addi    a0, a0, %lo(time)
            tail    printStr             ; Jump to string output
                                         ; routine.
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 enable you to 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
```

For more information, see *Conditional assembly directives*, page 94.

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 absolute and relocatable expressions as follows:

```
            name    simpleExpressions
            section MYCONST:CONST(2)
first       dc8     5                ; A relocatable label.
second      equ     10 + 5           ; An absolute expression.

            dc8     first            ; Examples of some legal
            dc8     first + 1        ; relocatable expressions.
            dc8     first + second
            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 can be absolute or relative depending on the type of section. 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 11: Symbol and cross-reference table

Programming hints

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

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for several RISC-V devices are included in the IAR Systems product package, in the `riscv\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 RISC-V, and they are suitable to use as templates when creating new header files for other RISC-V derivatives.

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 114.

C-style preprocessor directives like `#define` are valid in the remainder of the source code file, while assembler directives like `EQU` only are valid in the current module.

Tracking call frame usage

In this section, these topics are described:

- *Call frame information overview*, page 30
- *Call frame information in more detail*, page 30

These tasks are described:

- *Defining a names block*, page 31
- *Defining a common block*, page 32
- *Annotating your source code within a data block*, page 32
- *Specifying rules for tracking resources and the stack depth*, page 33
- *Using CFI expressions for tracking complex cases*, page 35
- *Stack usage analysis directives*, page 36
- *Examples of using CFI directives*, page 36

For reference information, see:

- *Call frame information directives for names blocks*, page 121
- *Call frame information directives for common blocks*, page 122
- *Call frame information directives for data blocks*, page 123
- *Call frame information directives for tracking resources and CFAs*, page 124
- *Call frame information directives for stack usage analysis*, page 127

CALL FRAME INFORMATION OVERVIEW

Call frame information (CFI) is information about the *call frames*. Typically, a call frame contains a return address, function arguments, saved register values, compiler temporaries, and local variables. Call frame information holds enough information about call frames to support two important features:

- C-SPY can use call frame information to reconstruct the entire call chain from the current `PC` (program counter) and show the values of local variables in each function in the call chain. This information is used, for example, in the **Call Stack** window.
- Call frame information can be used, together with information about possible calls for calculating the total stack usage in the application. Note that this feature might not be supported by the product you are using.

The compiler automatically generates call frame information for all C and C++ source code. Call frame information is also typically provided for each assembler routine in the system library. However, if you have other assembler routines and want to enable C-SPY to show the call stack when executing these routines, you must add the required call frame information annotations to your assembler source code. Stack usage can also be handled this way (by adding the required annotations for each function call), but you can also specify stack usage information for any routines in a *stack usage control file* (see the *IAR C/C++ Development Guide for RISC-V*), which is typically easier.

CALL FRAME INFORMATION IN MORE DETAIL

You can add call frame information to assembler files by using `cfi` directives. You can use these to specify:

- The *start address* of the call frame, which is referred to as the *canonical frame address* (CFA). There are two different types of call frames:
  - On a stack—*stack frames*. For stack frames the CFA is typically the value of the stack pointer after the return from the routine.
  - In static memory, as used in a static overlay system—*static overlay frames*. This type of call frame is not required by RISC-V and is therefore not supported.
- How to find the return address.
- How to restore various resources, like registers, when returning from the routine.

When adding the call frame information for each assembler module, you must:

1. Provide a *names block* where you describe the resources to be tracked.
2. Provide a *common block* where you define the resources to be tracked and specify their default values. This information must correspond to the calling convention used by the compiler.

3 Annotate the resources used in your source code, which in practice means that you describe the changes performed on the call frame. Typically, this includes information about when the stack pointer is changed, and when permanent registers are stored or restored on the stack.

   To do this you must define a *data block* that encloses a continuous piece of source code where you specify *rules* for each resource to be tracked. When the descriptive power of the rules is not enough, you can instead use *CFI expressions*.

A full description of the calling convention might require extensive call frame information. In many cases, a more limited approach will suffice. The recommended way to create an assembler language routine that handles call frame information correctly is to start with a C skeleton function that you compile to generate assembler output. For an example, see the *IAR C/C++ Development Guide for RISC-V*.

DEFINING A NAMES BLOCK

A *names block* is used for declaring 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, and 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. The name must be one of the register names defined in the RISC-V ABI specification. 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 memory 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.

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 memory in which the calling function resides. You must declare the return address column for the common block.

Inside a common block, you can declare the initial value of a CFA or a resource by using the directives available for common blocks, see *Call frame information directives for common blocks*, page 122. For more information about how to use these directives, see *Specifying rules for tracking resources and the stack depth*, page 33 and *Using CFI expressions for tracking complex cases*, page 35.

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

ANNOTATING YOUR SOURCE CODE WITHIN A DATA BLOCK

The *data block* contains the actual tracking information for one continuous piece of code.

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 for the current data block 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 for the current data block 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 resources by using the directives available for data blocks, see *Call frame information directives for data blocks*, page 123. For more information on how to use these directives, see *Specifying rules for tracking resources and the stack depth*, page 33, and *Using CFI expressions for tracking complex cases*, page 35.

SPECIFYING RULES FOR TRACKING RESOURCES AND THE STACK DEPTH

To describe the tracking information for individual resources, two sets of simple rules with specialized syntax can be used:

- Rules for tracking resources

  ```
  CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
  CFI resource { resource | FRAME(cfa, offset) }
  ```

- Rules for tracking the stack depth (CFAs)

  ```
  CFI cfa { NOTUSED | USED }
  CFI cfa { resource | resource + constant | resource - constant }
  ```

You can use these 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* with dedicated *operators* to describe the information, see *Using CFI expressions for tracking complex cases*, page 35. However, whenever possible, you should always use a rule instead of a CFI expression.

Rules for tracking 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, in other words, already correctly located, use `SAMEVALUE` as the location. Conceptually, this declares that the resource does not have to be restored because it already contains the correct value. For example, to declare that a register `R11` is restored to the same value, use the directive:

```
CFI R11 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) because 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 `R11` is a scratch register and does not have to be restored, use the directive:

```
CFI R11 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 `R11` is temporarily located in a register `R12` (and should be restored from that register), use the directive:

```
CFI R11 R12
```

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 `R11` is located at offset –4 counting from the frame pointer `CFA_SP`, use the directive:

```
CFI R11 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.

Rules for tracking the stack depth (CFAs)

In contrast to 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 assembler call instruction. The CFA rules describe how to compute the address of the beginning of the current stack frame.

Each stack frame CFA is associated with a stack pointer. When going back one call frame, the associated stack pointer is restored to the current CFA. For stack frame CFAs there are two possible 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 stack pointer 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 stack pointer 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
```

USING CFI EXPRESSIONS FOR TRACKING COMPLEX CASES

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

CFI expressions consist of operands and operators. Three sets of operators are allowed in a CFI expression:

- Unary operators
- Binary operators
- Ternary operators

In most cases, they have an equivalent operator in the regular assembler expressions.

In this example, `A0` is restored to its original value. However, instead of saving it, the effect of the two post increments is undone by the subtract instruction.

```
AddTwo:
        cfi     block addTwoBlock using myCommon
        cfi     function addTwo
        cfi     nocalls
        cfi     a0 samevalue
        lw      a1, 0(a0)
        addi    a0, a0, 4
        cfi     a0 sub(a0, 4)
        lw      a2, 0(a0)
        addi    a0, a0, 4
        cfi     a0 sub(a0, 8)

        add     a1, a1, a2

        addi    a0, a0, -8
        cfi     a0 samevalue
        ret
        cfi     endblock addTwoBlock
        end
```

For more information about the syntax for using the operators in CFI expressions, see *Call frame information directives for tracking resources and CFAs*, page 124.

STACK USAGE ANALYSIS DIRECTIVES

The stack usage analysis directives (`CFI FUNCALL`, `CFI TAILCALL`, `CFI INDIRECTCALL`, and `CFI NOCALLS`) are used for building a call graph which is needed for stack usage analysis. These directives can be used only in data blocks. When the data block is a function block (in other words, when the `CFI FUNCTION` directive has been used in the data block), you should not specify a *`caller`* parameter. When a stack usage analysis directive is used in code that is shared between functions, you must use the *`caller`* parameter to specify which of the possible functions the information applies to.

The `CFI FUNCALL`, `CFI TAILCALL`, and `CFI INDIRECTCALL` directives must be placed immediately before the instruction that performs the call. The `CFI NOCALLS` directive can be placed anywhere in the data block.

EXAMPLES OF USING CFI DIRECTIVES

The following is a generic example of how to add and use the required CFI directives. The example is not specific to RISC-V. To obtain an example specific to the core you are using, 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 may be destroyed by a function call—whereas register `R1` must be restored after the function call. To simplify, all instructions, registers, and addresses are assumed to have a width of 16 bits.

Consider the following short code example with the corresponding call frame information. 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, in other words, the value of the stack pointer after returning from the function.

Address	CFA	R0	R1	RET	Assembler code
0000	SP + 2	undefined	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 12: Code sample with call frame information

Each 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 row at address `0000` is the initial row and the result of the calling convention used for the function.

The RET column is the return address column—that is, the location of the return address. The value of `R0` is undefined because it 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

            ; Offset -2 from top of frame.
            cfi  RET frame(CFA,-2)
            cfi     endcommon trivialCommon
```

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

Annotating your source code within a data block

You should place the `CFI` directives at the point where the call frame information has changed, in other words, immediately *after* the instruction that changes the call frame information.

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
```

Assembler options

- Using command line assembler options
- Summary of assembler options
- Description of assembler options

Using command line assembler options

Assembler options are parameters you can specify to change the default behavior of the assembler. You can specify options from the command line—which is described in more detail in this section—and from within the IAR Embedded Workbench® IDE.

The *IDE Project Management and Building Guide for RISC-V* describes how to set assembler options in the IDE, and gives reference information about the available options.

SPECIFYING COMMAND LINE OPTIONS

To set assembler options from the command line, include them on the command line after the `iasmriscv` 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:

```
iasmriscv prog.s --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`:

```
iasmriscv prog.s -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:

```
iasmriscv prog.s -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, with or without 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 slash or backslash can be added to the last directory name, but is not required.

Additionally, some 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:

```
iasmriscv 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:

```
iasmriscv prog -l ---
```

EXTENDED COMMAND LINE FILE

In addition to accepting options and source filenames from the command line, the assembler can accept them from an extended command line file.

By default, extended command line files have the extension `xcl`, and can be specified using the `-f` command line option. For example, to read the command line options from `extend.xcl`, enter:

```
iasmriscv -f extend.xcl
```

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
`--core`	Specifies which processor core extensions the code will be generated for
`-D`	Defines preprocessor symbols
`--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
`--error_limit`	Specifies the allowed number of errors before the assembler stops
`-f`	Extends the command line
`--f`	Extends the command line, optionally with a dependency
`--header_context`	Lists all referred source files
`-I`	Adds a search path for a header file
`-l`	Generates a list file
`-M`	Macro quote characters
`--macro_positions_in_diagnostics`	Obtains positions inside macros in diagnostic messages
`--mnem_first`	Allows mnemonics in the first column
`--no_bom`	Omits the Byte Order Mark for UTF-8 output files

Table 13: Assembler options summary

Command line option	Description
--no_call_frame_info	Disables output of call frame information
--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
--nonportable_path_warnings	Generates a warning when the path used for opening a source header file is not in the same case as the path in the file system.
-o	Sets the object filename. Alias for --output.
--only_stdout	Uses standard output only
--output	Sets the object filename
--predef_macros	Lists the predefined symbols
--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
--source_encoding	Specifies the encoding for source files
--system_include_dir	Specifies the path for system include files
--text_out	Specifies the encoding for text output files
--use_paths_as_written	Use paths as written in debug information
--use_unix_directory_separators	Uses / as directory separator in paths
--utf8_text_in	Uses the UTF-8 encoding for text input files
--version	Sends assembler output to the console and then exits.
--warnings_affect_exit_code	Warnings affect exit code
--warnings_are_errors	Treats all warnings as errors

Table 13: Assembler options summary (Continued)

Description of assembler options

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

If you use the page **Extra Options** to specify specific command line options, the IDE does not perform an instant check for consistency problems like conflicting options, duplication of options, or use of irrelevant options.

--case_insensitive

Syntax

`--case_insensitive`

Description

Use this option to make user symbols case-insensitive. By default, case sensitivity is on.

You can also use the assembler directives `CASEON` and `CASEOFF` to control case sensitivity for user-defined symbols.

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.

Example

By default, for example, `LABEL` and `label` refer to different symbols. When `--case_insensitive` is used, `LABEL` and `label` instead refer to the same symbol.

See also

Assembler control directives, page 114 and information about defining and undefining preprocessor symbols under *C-style preprocessor directives*, page 106.

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

--core

Syntax

`--core=RV32{IM|G|EM}[A][F][D][C]`

Parameters

`RV32`	Generates code for 32-bit RISC-V devices
`EM`	Supports the RV32E Base Integer Instruction Set and the M extension
`G`	Supports the RV32I Base Integer Instruction Set and the M, A, F, and D extensions.
`IM`	Supports the RV32I Base Integer Instruction Set and the M extension

`A`	Supports the A extension
`F`	Supports the F extension
`D`	Supports the D extension
`C`	Supports the C extension

Description

Use this option to specify which processor core extensions the code will be generated for. Code that requires extensions that you have not specified using this option, will be rejected by the assembler. Note that all modules of your application must use the same parameters.

Project>Options>General Options>Target>Device

-D

Syntax

`-D`*`symbol`*`[=`*`value`*`]`

Parameters

`symbol`	The name of the symbol you want to define.
`value`	The value of the symbol. If no value is specified, `1` is used.

Description

Use this option to define a symbol to be used by the preprocessor.

Example

You might want to arrange your source code to produce either the test version or the production version of your application, depending 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:	`iasmrisc-v prog`
Test version:	`iasmrisc-v 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:

```
iasmrisc-v prog -DFRAMERATE=3
```

Project>Options>Assembler>Preprocessor>Defined symbols

--debug, -r

Syntax

```
--debug
-r
```

Description

Use this option to make the assembler generate debug information, which means the generated output can be used in a symbolic debugger such as IAR C-SPY® Debugger.

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

Syntax

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

Parameters

No parameter	The same affect as for the parameter `i`.

`i` (default)	The names of the dependent files, including the full path if available, is output. For example:
	`c:\iar\product\include\stdio.h`
	`d:\myproject\include\foo.h`

`m`	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`

`filename`	The output is stored in the specified file.

directory	The output is stored in a file (filename extension `i`) which is stored in the specified directory.

For information about specifying a filename or directory, see *Specifying parameters*, page 40.

Description

Use this option to list each source file opened by the assembler in a file.

Example

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

```
iasmriscv prog --dependencies=i listing
```

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, in addition to 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

Syntax

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

Parameters

tag	The number of a diagnostic message, for example the message number `As001`.

Description

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. The option can be used more than once on the command line.

Example

This example classifies warning `As001` as an error:

```
--diag_error=As001
```

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

--diag_remark

Syntax

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

Parameters

tag	The number of a diagnostic message, for example the message number `As001`.

Description

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.

Example

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

```
--diag_remark=As001
```

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

--diag_suppress

Syntax

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

Parameters

tag	The number of a diagnostic message, for example the message number `As001`.

Description

Use this option to suppress diagnostic messages.

Example

This example suppresses the warnings `As001` and `As002`:

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

Project>Options>Assembler >Diagnostics>Suppress these diagnostics

--diag_warning

Syntax

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

Parameters

`tag`	The number of a diagnostic message, for example the message number `As001`.

Description

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.

Example

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

```
--diag_warning=As028
```

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

--diagnostics_tables

Syntax

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

Parameters

`filename`	The diagnostic messages are stored in the specified file.
`directory`	The diagnostic messages are stored in a file (filename extension `i`) which is stored in the specified directory.

For information about specifying a filename or directory, see *Specifying parameters*, page 40.

Description

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.

Example

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

```
--diagnostics_tables diag
```

This option is not available in the IDE.

--dir_first

Syntax

`--dir_first`

Description

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

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

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

--error_limit

Syntax

`--error_limit=`*`n`*

Parameters

`n`	The number of errors before the assembler stops the assembly. *`n`* must be a positive integer; `0` indicates no limit.

Description

Use this option to specify the number of errors allowed before the assembler stops. By default, 100 errors are allowed.

This option is not available in the IDE.

-f

Syntax

`-f` *`filename`*

Parameters

`filename`	The commands that you want to extend the command line with are read from the specified file. Notice that there must be a space between the option itself and the filename.

For information about specifying a filename, see *Specifying parameters*, page 40.

Description

Use this option to extend the command line with text read from the specified file.

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:

```
iasmriscv prog -f extend.xcl
```

See also

--f, page 50 and *Extended command line file*, page 40.

To set this option, use:

Project>Options>Assembler>Extra Options

--f

Syntax

```
--f filename
```

Parameters

filename	The commands that you want to extend the command line with are read from the specified file. Notice that there must be a space between the option itself and the filename.

For information about specifying a filename, see *Specifying parameters*, page 40.

Description

Use this option to make the assembler read command line options from the named file, with the default filename extension `xcl`.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

If you use the assembler option `--dependencies`, extended command line files specified using `--f` will generate a dependency, but those specified using `-f` will not generate a dependency.

See also

--dependencies, page 45 and *-f*, page 49.

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

--header_context

Syntax

`--header_context`

Description

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

Syntax

`-I`*`path`*

Parameters

`path`	The search path for `#include` files.

Description

Use this option to specify paths to be used by the preprocessor. This option can be used more than once on the command line.

By default, the assembler searches for `#include` files in the current working directory, in the system header directories, and in the paths specified in the `IASMrisc-v_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 code, make 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 `IASMrisc-v_INC` environment variable, provided that this variable is set, and in the system header directories.

Project>Options>Assembler>Preprocessor>Additional include directories

-l

Syntax

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

Parameters

`a`	Assembled lines only.
`d`	The `LSTOUT` directive controls if lines are written to the list file or not. Using `-ld` turns the start value for this to off.
`e`	No macro expansions.
`m`	Macro definitions.
`o`	Multiline code.
`x`	Includes cross-references.
`N`	Do not include diagnostics.
`H`	Includes header file source lines.
`filename`	The output is stored in the specified file.
`directory`	The output is stored in a file (filename extension `i`) which is stored in the specified directory.

For information about specifying a filename or directory, see *Specifying parameters*, page 40.

Description

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

Example

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

```
iasm sourcefile -l list
```

To set related options, select:

Project>Options>Assembler >List

-M

Syntax

`-Mab`

Parameters

`ab`	The characters to be used as left and right quotes of each macro argument, respectively.

Description

Use this option to 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 allows 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:

`iasmriscv` *`filename`* `-M'<>'`

Project>Options>Assembler >Language>Macro quote characters

--macro_positions_in_diagnostics

Syntax

```
--macro_positions_in_diagnostics
```

Description

Use this option to obtain position references inside macros in diagnostic messages. This is useful for detecting incorrect source code constructs in macros.

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

--mnem_first

Syntax

```
--mnem_first
```

Description

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

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

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

--no_bom

Syntax

`--no_bom`

Description

Use this option to omit the Byte Order Mark (BOM) when generating a UTF-8 output file.

See also

--text_out, page 59. For more information about encodings, see the *IAR C/C++ Development Guide for RISC-V*.

Project>Options>Assembler>Encodings>Text output file encoding

--no_call_frame_info

Syntax

`--no_call_frame_info`

Description

By default, the assembler generates call frame information in object files for assembler code that is annotated with such information (but only for such code). Use this option to disable the generation of call frame information entirely.

See also

Tracking call frame usage, page 29.

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

--no_path_in_file_macros

Syntax

`--no_path_in_file_macros`

Description

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

Syntax

```
--no_system_include
```

Description

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

Syntax

```
--no_warnings
```

Description

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

Syntax

```
--no_wrap_diagnostics
```

Description

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.

--nonportable_path_warnings

Syntax

```
--nonportable_path_warnings
```

Description

Use this option to make the assembler generate a warning when characters in the path used for opening a source file or header file are lower case instead of upper case, or vice versa, compared with the path in the file system.

This option is not available in the IDE.

--only_stdout

Syntax

`--only_stdout`

Description

Use this option to make the assembler direct messages to `stdout` instead of to `stderr`.

This option is not available in the IDE.

--output, -o

Syntax

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

Parameters

filename	The object code is stored in the specified file.
directory	The object code is stored in a file (filename extension `o`) which is stored in the specified directory.

For information about specifying a filename or directory, see *Specifying parameters*, page 40.

Description

By default, the object code 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 specify a different output filename for the object code output.

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

--predef_macros

Syntax

`--predef_macros {filename|directory}`

Parameters

filename	The list of predefined macros is stored in the specified file.
directory	The list of predefined macros is stored in a file (filename extension `predef`) which is stored in the specified directory.

For information about specifying a filename or directory, see *Specifying parameters*, page 40.

Description

Use this option to list all symbols defined by the assembler or on the command line. When using this option, make sure to also use the same options as for the rest of your project.

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

This option is not available in the IDE.

--preinclude

Syntax

`--preinclude` *`includefile`*

Parameters

`includefile`	The header file to be included.

Description

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>Preprocessor>Preinclude file

--preprocess

Syntax

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

Parameters

No parameter	A preprocessed file.
`c`	Preserves C and C++ style comments that otherwise are removed by the preprocessor. Assembler style comments are always preserved.
`n`	Preprocess only.
`s`	Suppress `#line` directives.
`filename`	The output is stored in the specified file.
`directory`	The output is stored in a file (filename extension `i`) which is stored in the specified directory. The filename is the same as the name of the assembled source file.

For information about specifying a filename or directory, see *Specifying parameters*, page 40.

Description

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

Example

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

```
iasmriscv sourcefile --preprocess=c output
```

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

--remarks

Syntax

```
--remarks
```

Description

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 also

Severity levels, page 133.

Project>Options>Assembler >Diagnostics>Enable remarks

--silent

Syntax

```
--silent
```

Description

By default, the assembler sends various minor messages via the standard output stream. Use this option to make the assembler operate without sending any messages to the standard output stream.

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.

--source_encoding

Syntax
```
--source_encoding {locale|utf8}
```

Parameters

`locale`	The default source encoding is the system locale encoding.
`utf8`	The default source encoding is the UTF-8 encoding.

Description

When reading a source file with no Byte Order Mark (BOM), use this option to specify the encoding. If this option is not specified and the source file does not have a BOM, the Raw encoding will be used.

See also

For more information about encodings, see the *IAR C/C++ Development Guide for RISC-V*.

Project>Options>Assembler>Encodings>Default source file encoding

--system_include_dir

Syntax
```
--system_include_dir path
```

Parameters

path	The path to the system include files.

Description

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.

--text_out

Syntax
```
--text_out {utf8|utf16le|utf16be|locale}
```

Parameters

`utf8`	Uses the UTF-8 encoding
`utf16le`	Uses the UTF-16 little-endian encoding
`utf16be`	Uses the UTF-16 big-endian encoding

`locale`	Uses the system locale encoding

Description

Use this option to specify the encoding to be used when generating a text output file.

The default for the assembler list files is to use the same encoding as the main source file. The default for all other text files is UTF-8 with a Byte Order Mark (BOM).

If you want text output in UTF-8 encoding without a BOM, use the option `--no_bom`.

See also

--*no_bom*, page 54. For more information about encodings, see the *IAR C/C++ Development Guide for RISC-V*.

Project>Options>Assembler>Encodings>Text output file encoding

--use_paths_as_written

Syntax

`--use_paths_as_written`

Description

By default, the assembler ensures that all paths in the debug information are absolute, even if not originally specified that way.

If you use this option, paths that were originally specified as relative will be relative in the debug information.

The paths affected by this option are:

- the paths to source files
- the paths to header files that are found using an include path that was specified as relative

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

--use_unix_directory_separators

Syntax

`--use_unix_directory_separators`

Description

Use this option to make DWARF debug information use `/` (instead of `\`) as directory separators in file paths.

This option can be useful if you have a debugger that requires directory separators in UNIX style.

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

--utf8_text_in

Syntax

`--utf8_text_in`

Description

Use this option to specify that the assembler shall use UTF-8 encoding when reading a text input file with no Byte Order Mark (BOM).

Note: This option does not apply to source files.

See also

The *IAR C/C++ Development Guide for RISC-V* for more information about encodings.

Project>Options>Assembler>Encodings>Default input file encoding

--version

Syntax

`--version`

Description

Use this option to make the assembler send version information to the console and then exit.

This option is not available in the IDE.

--warnings_affect_exit_code

Syntax

`--warnings_affect_exit_code`

Description

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

This option is not available in the IDE.

--warnings_are_errors

Syntax

```
--warnings_are_errors
```

Description

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
```

See also

--*diag_warning*, page 48.

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

Assembler operators

- Precedence of assembler operators
- Summary of assembler operators
- Description of assembler operators

Precedence of assembler 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))
  ```

Summary of assembler operators

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

PARENTHESIS OPERATOR

Precedence: 1

`()`	Parenthesis.

FUNCTION OPERATORS

Precedence: 2

`%hi`	Upper 20 bits, compensated and shifted
`%lo`	Lower 12 bits, sign extended
`%pcrel_hi`	Upper 20 bits, PC-relative, compensated and shifted
`%pcrel_lo`	Lower 12 bits, PC-relative and sign extended
`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

UNARY OPERATORS

Precedence: 3

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

MULTIPLICATIVE ARITHMETIC OPERATORS

Precedence: 4

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

ADDITIVE ARITHMETIC OPERATORS

Precedence: 5

`+`	Addition
`–`	Subtraction

SHIFT OPERATORS

Precedence: 6

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

COMPARISON OPERATORS

Precedence: 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

Precedence: 8

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

LOGICAL OPERATORS

Precedence: 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

Precedence: 15

`? :`	Conditional operator

Description of assembler operators

This section gives detailed descriptions of each assembler operator.

See also *Expressions, operands, and operators*, page 20.

() Parenthesis

Precedence
1

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

Example

```
1+2*3 -> 7
(1+2)*3 -> 9
```

* Multiplication

Precedence
4

Description
`*` 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

Precedence 3

Description Unary plus operator; performs nothing.

Example

```
+3 -> 3
3*+2 -> 6
```

+ Addition

Precedence 5

Description 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

Precedence 3

Description 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

Precedence

5

Description

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

Precedence

4

Description

`/` produces the integer quotient of the left operand divided by the right operator. 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

Syntax

condition ? *expr* : *expr*

Precedence

15

Description

`?` results in 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.

Example

```
5 ? 6 : 7 ->6
0 ? 6 : 7 ->7
```

< Less than

Precedence 7

Description `<` or `LT` evaluates to 1 (true) if the left operand has a numeric value that is less than the right operand, otherwise it is 0 (false).

Example

```
-1 < 2 -> 1
2 < 1 -> 0
2 < 2 -> 0
```

<= Less than or equal to

Precedence 7

Description `<=` or `LE` evaluates to 1 (true) if the left operand has a numeric value that is less than or equal to the right operand, otherwise it is 0 (false).

Example

```
1 <= 2 -> 1
2 <= 1 -> 0
1 <= 1 -> 1
```

<>, != Not equal to

Precedence 8

Description `<>`, `!=`, or `NE` 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
```

=, == Equal to

Precedence 8

Description `=`, `==`, or `EQ` 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
```

> Greater than

Precedence 7

Description

`>` or `GT` 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
```

>= Greater than or equal to

Precedence 7

Description

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

Example

```
1 >= 2 -> 0
2 >= 1 -> 1
1 >= 1 -> 1
```

&& Logical AND

Precedence 12

Description

`&&` or `AND` performs 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 && 0011B -> 1
1010B && 0101B -> 1
1010B && 0000B -> 0
```

& Bitwise AND

Precedence

9

Description

`&` or `BINAND` or `BITAND` performs 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 & 0011B -> 0010B
1010B & 0101B -> 0000B
1010B & 0000B -> 0000B
```

~ Bitwise NOT

Precedence

3

Description

`~` or `BINNOT` or `BITNOT` performs bitwise NOT on its operand. Each bit in the 32-bit result is the complement of the corresponding bit in the operand.

Example

```
~ 1010B -> 11111111111111111111111111110101B
```

| Bitwise OR

Precedence

11

Description

`|` or `BINOR` or `BITOR` performs 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 | 0101B -> 1111B
1010B | 0000B -> 1010B
```

^ Bitwise exclusive OR

Precedence

10

Description

`^` or `BINXOR` or `BITXOR` performs 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 ^ 0101B -> 1111B
1010B ^ 0011B -> 1001B
```

% Modulo

Precedence 4

Description

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

Example

```
2 % 2 -> 0
12 % 7 -> 5
3 % 2 -> 1
```

! Logical NOT

Precedence 3

Description

`!` or `NOT` negates a logical argument.

Example

```
! 0101B -> 0
! 0000B -> 1
```

|| Logical OR

Precedence 14

Description

`||` or `OR` performs a logical OR between two integer operands.

Example

```
1010B || 0000B -> 1
0000B || 0000B -> 0
```

<< Logical shift left

Precedence 6

Description

`<<` or `SHL` shifts 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 << 3 -> 11100000B
0000011111111111B << 5 -> 1111111111100000B
14 << 1 -> 28
```

>> Logical shift right

Precedence 6

Description `>>` or `SHR` shifts 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 >> 3 -> 00001110B
1111111111111111B >> 20 -> 0
14 >> 1 -> 7
```

%hi Upper 20 bits

Precedence 2

Description `%hi` takes the upper 20 bits, compensated for a negative lower 12-bit field, and right-shifted 12 positions. The intended use for this operator is for the instruction `LUI` and for instructions like `ADDI`, `LW`, and `SW` to construct a 32-bit address.

Example

```
lui     a0, %hi(myVar)
lw      a0, %lo(myVar)(a0)
```

%lo Lower 12 bits

Precedence 2

Description `%lo` takes the lower 12 bits and sign-extends them. The intended use for this operator is for instructions like `ADDI`, `LW`, and `SW` to construct a 32-bit value together with the `LUI` instruction.

Example

```
lui     a0, %hi(myVar)
addi    a0, a0, %lo(myVar)
```

%pcrel_hi Upper 20 bits PC-relative

Precedence
2

Description
`%pcrel_hi` uses PC-relative addressing and takes the upper 20 bits, compensated for a negative lower 12-bit field, right-shifted 12 positions. The intended use for this operator is for the instruction `AUIPC` and for instructions like `ADDI`, `LW`, and `SW` to construct a 32-bit address.

Example

```
auipc   a0, %pcrel_hi(myVar)
lw      a0, %pcrel_lo(myVar)(a0)
```

%pcrel_lo Lower 12 bits PC-relative

Precedence
2

Description
`%pcrel_lo` uses PC-relative addressing and takes the lower 12 bits, and sign-extends them. The intended use for this operator is for instructions like `ADDI`, `LW`, and `SW` to construct a 32-bit value together with the `AUIPC` instruction.

Example

```
auipc   a0, %pcrel_hi(myVar)
addi    a0, a0, %pcrel_lo(myVar)
```

BYTE1 First byte

Precedence
2

Description
`BYTE1` 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

```
BYTE1 0xABCD -> 0xCD
```

BYTE2 Second byte

Precedence
2

Description
`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

Precedence 2

Description `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

Precedence 2

Description `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 time/date

Precedence 2

Description `DATE` gets the time 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 specify the date of assembly:

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

HIGH High byte

Precedence 2

Description `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

Precedence 2

Description `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`

LOW Low byte

Precedence 2

Description `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`

LWRD Low word

Precedence 2

Description `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`

SFB section begin

Syntax

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

Precedence

2

Parameters

section	The name of a 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.

Description

`SFB` accepts a single operand to its right. The operator evaluates to the absolute address of the first byte of that section. This evaluation occurs at linking time.

Example

```
            name    sectionBegin
            section MYCODE:CODE(2)  ; Forward declaration
                                    ; of MYCODE.
            section MYCONST:CONST(2)
start       dc32    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

Syntax

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

Precedence

2

Parameters

section	The name of a 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.

Description

`SFE` accepts a single operand to its right. The operator evaluates to the address of the first byte after the section end. This evaluation occurs at linking time.

Example

```
                name    sectionEnd
                section MYCODE:CODE(2)   ; Forward declaration
                                         ; of MYCODE.
                section MYCONST:CONST(2)
end             dc32    sfe(MYCODE)
                end
```

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

The size of the section `MYCODE` can be achieved by using the `SIZEOF` operator.

SIZEOF section size

Syntax

`SIZEOF` *`section`*

Precedence

2

Parameters

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

Description

`SIZEOF` generates `SFE-SFB` for its argument. That is, it calculates the size in bytes of a section. This is done when modules are linked together.

Example

These two files set `size` to the size of the section `MYCODE`.

`Table.s`:

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

`Application.s`:

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

UGT Unsigned greater than

Precedence
7

Description
`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 the operands as unsigned values.

Example

```
2 UGT 1 -> 1
-1 UGT 1 -> 1
```

ULT Unsigned less than

Precedence
7

Description
`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
```

XOR Logical exclusive OR

Precedence
13

Description
`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 a 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 85
- *Symbol control directives*, page 87
- *Section control directives*, page 89
- *Value assignment directives*, page 92
- *Conditional assembly directives*, page 94
- *Macro processing directives*, page 95
- *Listing control directives*, page 103
- *C-style preprocessor directives*, page 106
- *Data definition or allocation directives*, page 111
- *Assembler control directives*, page 114
- *Custom instruction directives*, page 117
- *Function directives*, page 120
- *Call frame information directives for names blocks*, page 121.
- *Call frame information directives for common blocks*, page 122
- *Call frame information directives for data blocks*, page 123
- *Call frame information directives for tracking resources and CFAs*, page 124
- *Call frame information directives for stack usage analysis*, page 127

This table gives a summary of all the assembler directives:

Directive	Description	Section
`#define`	Assigns a value to a label.	C-style preprocessor
`#elif`	Introduces a new condition in an `#if`…`#endif` block.	C-style preprocessor
`#else`	Assembles instructions if a condition is false.	C-style preprocessor
`#endif`	Ends an `#if`, `#ifdef`, or `#ifndef` block.	C-style preprocessor

Table 14: Assembler directives summary

Directive	Description	Section
#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
#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
.insn	Generates instructions not natively supported by the IAR Assembler.	Custom instruction
.option	Makes a setting that controls the operation of the assembler.	Assembler control
/*comment*/	C-style comment delimiter.	Assembler control
//	C++ style comment delimiter.	Assembler control
=	Assigns a permanent value local to a module.	Value assignment
_args	Is set to number of arguments passed to macro.	Macro processing
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
CALL_GRAPH_ROOT	Specifies that a function is a call graph root.	Function
CASEOFF	Disables case sensitivity.	Assembler control
CASEON	Enables case sensitivity.	Assembler control
CFI	Specifies call frame information.	Call frame information
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 14: 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
`EXTWEAK`	Imports an external symbol (which can be undefined.	Symbol control

Table 14: Assembler directives summary (Continued)

Directive	Description	Section
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
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
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
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

Table 14: Assembler directives summary (Continued)

Directive	Description	Section
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 14: Assembler directives summary (Continued)

Description of assembler directives

The following pages give reference information about the assembler directives.

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.

Description

Module control directives are used for marking the beginning and end of source program modules, and for assigning names to them. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 27.

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; alias to PROGRAM.	No external references Absolute
PROGRAM	Begins a module.	No external references Absolute
RTMODEL	Declares runtime model attributes.	Not applicable

Table 15: Module control directives

Beginning a program module

Use NAME or PROGRAM to begin a program module, and to assign a name for future reference by the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian.

Program modules are unconditionally linked by XLINK, even if other modules do not reference them.

Beginning a module

Use any of the directives NAME or PROGRAM 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 RISC-V*.

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 RISC-V*.

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 "Platform", "M7"
; ...
end
```

Assembler source file `f2.s`:

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

Assembler source file `f3.s`:

```
module  mod_3
rtmodel "Platform", "M7"
; ...
end
```

Symbol control directives

Syntax

`EXTERN` *`symbol`* `[,`*`symbol`*`]` …

`EXTWEAK` *`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.

Description

These directives control how symbols are shared between modules:

Directive	Description
`EXTERN`, `IMPORT`	Imports an external symbol.

Table 16: Symbol control directives

Directive	Description
EXTWEAK	Imports an external symbol. The symbol can be undefined.
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 16: Symbol control directives (Continued)

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 in more than one module. 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.

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 even if the code is not referenced.

Example

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
            rseg    CODE:CODE

err         jal     a0, print
            dc8     "** Error **"
            ret

            end
```

Section control directives

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.
The default align value is 4.

`flag` `ROOT`, `NOROOT`

`ROOT` (the default mode) indicates that the section fragment must not be discarded.

`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.

	REORDER, NOREORDER NOREORDER (the default mode) starts a new fragment in the section with the given name, or a new section if no such section exists. REORDER starts a new section with the given name.
section	The name of the section. The section name is a user-defined symbol that follows the rules described in *Symbols*, page 22.
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.

Description

The section directives control how code and data are located. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 27.

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
ODD	Aligns the program counter to an odd address.	No external references Absolute
RSEG	Begins an ELF 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 17: Section control directives

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 `DC8` or `DS8`, 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.

In the following example, the data following the first `RSEG` directive is placed in a relocatable section called `TABLE`.

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

```
            module  calculate
            extern  operator
            extern  addOperator, subOperator

            rseg    TABLE:CONST(8)
operatorTable:
            dc32    addOperator, subOperator

            rseg    CODE:CODE
calculate   lui     a0, %hi(operator)
            lw      a0, %lo(operator)(a0)
            lui     a1, %hi(operatorTable)
            addi    a1, a1, %lo(operatorTable)
            lw      a2, 0(a1)
            beq     a0, a2, add
            lw      a2, 4(a1)
            beq     a0, a2, sub

            ;...
            ret

add         ;...
            ret
sub         ;...
            ret

            end
```

Aligning a section

Use `ALIGN` to align the program location counter to a specified address boundary. You do this by specifying an expression for the power of two to which the program counter should be aligned. That is, a value of 1 aligns to an even address and a value of 2 aligns to an address evenly divisible by 4.

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 value used for padding bytes must be within the range 0 to 255.

Use `ALIGNRAM` to align the program location counter by incrementing it; no data is generated. The parameter `align` can be within the range 0 to 31.

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
            rseg    MYDATA:CONST ; Start relocatable data section
            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

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.

Description

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 18: Value assignment directives

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.

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
```

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.

Conditional assembly directives

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.

Description

Use the `IF`, `ELSE`, `ELSEIF`, 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 `ELSEIF` condition is true or `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` and `ELSEIF` directives are optional, and if used, they must be inside an `IF...ENDIF` block. `IF...ENDIF` and `IF...ELSE...ENDIF` blocks can be nested to any level.

Example

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

```
; If the second argument to the setMem macro is 0, the macro
; writes register zero t0 to the memory location. For any other
; value of the second argument, the macro loads the value into t0
; and stores it.

setMem      macro   loc,val          ; loc is a direct page memory
                                     ; location, and val is an
                                     ; 8-bit value to add to that
                                     ; location.
            if      val = 0
            sw      zero, loc
            else
            addi    t0, zero, val
            sb      t0, loc
            endif
            endm


            module  addWithMacro
            rseg    CODE:CODE

addSome     setMem  0xa0(zero),0     ; Set 0 to memory loc. 0xa0.
            setMem  0xa0(zero),1     ; Set 1 to the same address.
            setMem  0xa0(zero),47    ; Set 47 to the same address.
            ret
            end
```

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

Description

These directives allow user macros to be defined. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 27.

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 19: Macro processing directives

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.

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.

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:

```
            section `.text`:CODE(2)

            name    errMacro
errMac      macro   text
            extern  abort
            call    abort
            dc8     text, 0
            even
            endm

            end
```

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:

```
            call   abort
            dc8    'Disk not ready', 0
            even
```

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:

```
            section `.text`:CODE(2)
            name    errMacro
errMac      macro   text
            extern  abort
            call    abort
            dc8     \1,0
            endm
            end
```

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:

```
            section `.text`:CODE(2)
            name    loadMac
loadMac     macro   ops
            lw      ops
            endm
            end
```

The macro can be called using the macro quote characters:

```
            loadMac <a0,0(a0)>
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 52.

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

            module  filler
            section `.text`:CODE(2)
            fill    3
            fill    4, 3
            end
```

It generates this code:

```
19    000000                            section `.text`:CODE(2)
20    000000                            fill    3
20.1  000000                            if      _args == 2
20.2  000000                            else
20.3  000000 03                         dc8     3
20.4  000001                            endif
21    000001                            fill    4, 3
21.1  000001                            if      _args == 2
21.2  000001                            rept    3
21.3  000001 04                         dc8     4
21.4  000002 04                         dc8     4
21.5  000003 04                         dc8     4
21.6  000004                            endr
21.7  000004                            else
21.8  000004                            endif
22
23    000004                            end
```

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.

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

```
            name    reptc
            extern  plotc
            rseg    CODE:CODE

banner      reptc   chr, "Welcome"
            addi    a0, x0, 'chr'
            call    plotc
            endr
            end
```

This produces this code:

```
10    000000                          extern  plotc
11    000000                          rseg    CODE:CODE
12
13    000000                banner    reptc   chr, "Welcome"
13.1  000000 0570'0513                addi    a0, x0, 'W'
13.2  000008 0000'80E7                call    plotc
13.3  00000C 0650'0513                addi    a0, x0, 'e'
13.4  000014 0000'80E7                call    plotc
13.5  000018 06C0'0513                addi    a0, x0, 'l'
13.6  000020 0000'80E7                call    plotc
13.7  000024 0630'0513                addi    a0, x0, 'c'
13.8  00002C 0000'80E7                call    plotc
13.9  000030 06F0'0513                addi    a0, x0, 'o'
13.10 000038 0000'80E7                call    plotc
13.11 00003C 06D0'0513                addi    a0, x0, 'm'
13.12 000044 0000'80E7                call    plotc
13.13 000048 0650'0513                addi    a0, x0, 'e'
13.14 000050 0000'80E7                call    plotc
13.15 000054                          endr
17    000054                          end
```

This example uses REPTI to clear several memory locations:

```
            name    repti
            extern  base, count, init
            rseg    CODE:CODE

banner      repti   adds, base, count, init
            lui     t0, %hi(adds)
            sw      x0, %lo(adds)(t0)
            endr

            end
```

This produces this code:

```
10    000000                         extern base, count, init
11    000000                         rseg   CODE:CODE
12
13    000000           banner    repti adds, base, count, init
13.1  000000 0000'02B7               lui    t0, %hi(base)
13.2  000004 0002'A023               sw     x0, %lo(base)(t0)
13.3  000008 0000'02B7               lui    t0, %hi(count)
13.4  00000C 0002'A023               sw     x0, %lo(count)(t0)
13.5  000010 0000'02B7               lui    t0, %hi(init)
13.6  000014 0002'A023               sw     x0, %lo(init)(t0)
13.7  000018                         endr
17
18    000018                         end
```

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:

```
             section `.text`:CODE(2)
             name    ioBufferSubroutine
             public  copyBuffer
ptbd         equ     0x0002          ; Definition of the port B
                                     ; data register.
             rseg    `.data`:DATA(0)
buffer       ds8     256
             rseg    `.text`:CODE(2)
copyBuffer   mv      t0, x0          ; Initialize the counter.
             addi    t1, x0, 256
             lui     a0, %hi(buffer)
loop         lbu     a1, %lo(buffer)(a0)
             sb      a1, ptbd(x0)
             addi    t0, t0, 1
             bne     t0, t1, loop    ; Have we copied 256 bytes?
             ret
             end
```

The main program calls this routine as follows:

```
doCopy       call    copyBuffer
```

For efficiency we can recode this using a macro:

```
             name    ioBufferInline
ptbd         equ     0x0002          ; Definition of the port B
                                     ; data register.
             rseg    `.data`:DATA(0)
buffer       ds8     256
copyBuffer   macro
             local   loop
             mv      t0, x0          ; Initialize the counter.
             addi    t1, x0, 256
             lui     a0, %hi(buffer)
loop         lbu     a1, %lo(buffer)(a0)
             sb      a1, ptbd(x0)
             addi    t0, t0, 1
             bne     t0, t1, loop    ; Have we copied 256 bytes?
             endm
             rseg    `.text`:CODE(2)
             copyBuffer
             ret
             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.

Listing control directives

Syntax

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

Description

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 20: 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.

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).

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

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.

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

```
            name    lstcndTest
            extern  print
            rseg    `.text`:CODE(2)
debug       set     0
begin       if      debug
            call    print
            endif
            lstcnd+
begin2      if      debug
            call    print
            endif
            end
```

This generates the following listing:

```
    10    000000                          extern  print
    11    000000                          rseg    `.text`:CODE(2)
    12
    13    000000             debug        set     0
    14    000000             begin        if      debug
    15                                    call    print
    16    000000                          endif
    17
    18                                    lstcnd+
    19    000000             begin2       if      debug
    21    000000                          endif
    22
    23    000000                          end
```

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.

This example shows the effect of LSTMAC and LSTEXP:

```
            name    lstmacTest
            rseg    FLASH:CODE

dec2        macro   arg
            addi    arg, arg, -1
            addi    arg, arg, -1
            endm

            lstmac+
inc2        macro   arg
            addi    arg, arg, 1
            addi    arg, arg, 1
            endm

begin       dec2    a0
            lstexp-
            inc2    a0
            ret

; Restore default values for
; listing control directives.

            lstmac-
            lstexp+

            end
```

This produces the following output:

```
10    000000                      rseg    FLASH:CODE
11
16
17                                lstmac+
18                         inc2   macro   arg
19                                addi    arg, arg, 1
20                                addi    arg, arg, 1
21                                endm
22
23    000000               begin  dec2    a0
23.1  000000 FFF5'0513            addi    a0, a0, -1
23.2  000004 FFF5'0513            addi    a0, a0, -1
24                                lstexp-
25    000008                      inc2    a0
26    000010 0000'8067            ret
27
28                         ; Restore default values for
29                         ; listing control directives.
30
31                                lstmac-
32                                lstexp+
33
34    000014                      end
```

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.

C-style preprocessor directives

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 assembler expression, see *Expressions, operands, and operators*, page 20. The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true. The C preprocessor operator `defined` can be used.
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.

Description

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.

Table 21: C-style preprocessor directives

Directive	Description
#line	Changes the source references in the debug information.
#pragma	Controls extension features. The supported #pragma directives are described in the chapter *Pragma directives*.
#undef	Undefines a preprocessor symbol.

Table 21: C-style preprocessor directives (Continued)

You should 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.

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
            rseg    CODE:CODE

#define     tweak  1
#define     adjust 3
calibrate   lui     t0, %hi(calibrationConstant)
            lw      t1, %lo(calibrationConstant)(t1)
#ifdef      tweak
#if         adjust==1
            addi    t1, t1, -4
#elif       adjust==2
            addi    t1, t1, -20
#elif       adjust==3
            addi    t1, t1, -30
#endif
#endif      /* ifdef tweak */
            sw      t1, %lo(calibrationConstant)(t1)
            ret

            end
```

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 <iodevice.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 Any directories specified using the `ARISCV_INC` environment variable.

  3 The automatically set up library system include directories. See --*no_system_include*, page 55 and --*system_include_dir*, page 59.

- 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 IAR Assembler for RISC-V, and double quotes for header files that are part of your application.

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

```
xch         macro   a,b
            addi    sp, sp, -16
            sw      \1, 0(sp)
            sw      \2, 4(sp)
            lw      \2, 0(sp)
            lw      \1, 4(sp)
            addi    sp, sp, 16
            endm
```

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

```
            program includeFile
            rseg    `.text`:CODE(2)

; Standard macro definitions.
#include "Macros.inc"

xchRegs     xch     a0, a1
            xch     t0, t1
            ret

            end
```

Displaying errors

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

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.

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. */

            module  misplacedComment2
            section MYCONST:CONST(2)

            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
```

Data definition or allocation directives

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

value A valid absolute expression or floating-point constant.

Description

These directives define values or reserve memory.

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.

For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 27.

The column *Alias* in the following table shows the null directive that corresponds to the IAR Systems directive.

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.
DF32	Generates 32-bit floating-point constants.
DF64	Generates 64-bit floating-point constants.
DQ15	Generates 16-bit fractional constants.

Table 22: Data definition or allocation directives

Directive	Description
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 22: Data definition or allocation directives (Continued)

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
            rseg    `.const`:CONST

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

            rseg    `.text`:CODE(2)
count       set     0

addTable    mv      a0, x0

            rept    7
            if      count == 7
            exitm
            endif
            lui     t0, %hi(table+count)
            lbu     t0, %lo(table+count)(t0)
            add     a0, a0, t0
count       set     count + 1
            endr

            ret

            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

Syntax

`/*`*`comment`*`*/`

`//`*`comment`*

`CASEOFF`

`CASEON`

`.option {norelax|pop|push|relax}`

`RADIX` *`expr`*

Parameters

`comment`	Comment ignored by the assembler.
`expr`	Default base; default 10 (decimal).
`norelax`	Disables the link-time instruction relaxation.
`pop`	Restores the options saved using `push`.
`push`	Saves the current options to a stack for future restoration.
`relax`	Enables the link-time instruction relaxation.

Description

These directives provide control over the operation of the assembler. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 27.

Directive	Description	Expression restrictions
`/*comment*/`	C-style comment delimiter.	
`//`	C++ style comment delimiter.	
`CASEOFF`	Disables case sensitivity.	
`CASEON`	Enables case sensitivity.	
`.option`	Makes a setting that controls the operation of the assembler.	
`RADIX`	Sets the default base on all numeric values.	No forward references No external references Absolute Fixed

Table 23: Assembler control directives

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

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

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.

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

```
            module  caseSensitivity1
            rseg    CODE:CODE

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

The following will generate a duplicate label error:

```
            module  caseSensitivity2
            rseg    CODE:CODE

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

Defining comments

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

```
/*
Program to read serial input.
Version 1: 19.2.11
Author: mjp
*/
```

See also *C-style preprocessor directives*, page 106.

Disabling and re-enabling link-time instruction relaxation

This example assumes that instruction relaxation is enabled initially. It first saves the current state of options. Then it disables instruction relaxation, executes two instructions, and after that restores instruction relaxation again:

```
.option push
.option norelax
la a0, myLabel  // will be a lui/addi pair, and not relaxed
                // to addi a0, gp, <offset>
call myFunc     // will be a auipc/jalr pair, even if a jal
                // would have reached myFunc
.option pop
```

For more information about instruction relaxation, see the *IAR C/C++ Development Guide for RISC-V*.

Changing the base

To set the default base to 16:

```
            module  radix
            rseg    CODE:CODE

            radix   16              ; With the default base set
            addi    t0, x0, 12      ; 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.
            addi    t0, x0, 12      ; Now, the immediate value of
            ;...                    ; the load instruction is
                                    ; interpreted as 0x0c.
            end
```

Custom instruction directives

Syntax `.insn format, {op2|op7}, operands`

Parameters

format The instruction format of the generated custom instruction. See the table *Custom assembler instructions* below.

op7 or *op2* An unsigned immediate value for either 2-bit or 7-bit operation code, depending on the instruction format.

operands Depending on the instruction format, these can be:

- *f2–f7*: unsigned immediate values for 2-bit to 7-bit function code
- *rd*, *rs1*, *rs2*, *rs3*: integer or floating-point destination and source registers (`x0–x31` or `f0–f31`, respectively)
- *rd'*, *rs1'*, *rs2'*: integer or floating-point compact instruction-constrained destination and source registers (`x8–x15` or `f8–f15`, respectively)
- *expr*: an immediate expression, whose width and sign depends on the instruction format. For some instruction formats, operators are allowed.

Description

These directives generate custom instructions not directly supported by the IAR Assembler for RISC-V. All RISC-V instruction formats are supported.

A custom instruction directive can be used with inline assembler in applications written in C or C++. In combination with an inline function, you can create an intrinsic-like function to use custom architecture extensions.

The bits for immediate values in compressed instructions (16-bit RVC) generally have an instruction-specific format that can differ even for instructions of the same type. All immediate values are copied directly into the bitfield—no rearrangements are performed, unless operators (like `%hi`) or relocations (see below) are used.

Some of the instructions allow relaxations to be performed by the linker. This can be disabled using the `.option norelax` directive.

These are the required operands for each instruction format:

Assembler instruction	Relaxed by linker	Resulting instruction format
`.insn b` *op7, f3, rd, rs1, expr*	—	Alias for `.insn sb`
`.insn ca` *op2, f6, f2, rd', rs2'*	—	CA
`.insn cb` *op2, f3, rs1', expr*	—	CB
`.insn ci` *op2, f2, rd, expr*	—	CI
`.insn ciw` *op2, f3, rd', expr*	—	CIW
`.insn cj` *op2, f3, expr*	—	CJ with `RISCV_RVC_JUMP` relocation
`.insn cr` *op2, f4, rd, rs1*	—	CR
`.insn cs` *op2, f3, rs1', rs2', expr*	—	CS
`.insn i` *op7, f3, rd, rs1, expr*	—	I, with relocations depending on *expr* (like `ADDI`)
`.insn i` *op7, f3, rd, expr*(*rs1*)	—	I, with relocations like `LW`
`.insn r` *op7, f3, f7, rd, rs1, rs2*	—	R
`.insn r` *op7, f3, f2, rd, rs1, rs2, rs3*	—	Alias for `.insn r4`
`.insn r4` *op7, f3, f2, rd, rs1, rs2, rs3*	—	R4
`.insn s` *op7, f3, rd, expr*(*rs1*)	If `%lo` is used	S, with relocations like `SW`
`.insn sb` *op7, f3, rd, rs1, expr*	—	B, with `RISCV_BRANCH` relocation
`.insn sb` *op7, f3, rd, expr*(*rs1*)	—	Alias for `.isns s`
`.insn u` *op7, f3, rd, expr*	If `%hi` is used	U, with relocations like `LUI` included
`.insn uj` *op2, rd, expr*	—	J, with `RISCV_JAL` relocation

Table 24: Custom assembler instructions

Refer to the RISC-V ISA specification, sections 2.3 and 12.2, for details on bit layout.

The operating code (*op2*/*op7*) can be supplied as an assembler constant expression, or as one of:

Operation code	**Value**
AMO	0x2f
AUIPC	0x17
BRANCH	0x63
C0	0x0
C1	0x1
C2	0x2
CUSTOM_0	0x0b
CUSTOM_1	0x2b
CUSTOM_2	0x5b
CUSTOM_3	0x7b
JAL	0x6f
JALR	0x67
LOAD	0x03
LOAD_FP	0x07
LUI	0x37
MADD	0x43
MISC_MEM	0x0f
MSUB	0x47
NMADD	0x4f
NMSUB	0x4b
OP	0x33
OP_32	0x3b
OP_FP	0x53
OP_IMM	0x13
OP_IMM_32	0x1b
STORE	0x23
STORE_FP	0x27
SYSTEM	0x73

Table 25: Constant value alternatives to opcodes

Examples

These lines of code show how to use these directives:

```
.insn i 0x13, 0x3, a0, a1, 0x40 // equivalent to
                                // sltiu a0, a1, 0x40
.insn s 0x23, 0, a0, 4(a1       // equivalent to sb a0, 4(a1)
.insn s STORE 0, a0, 4(a1)      // equivalent to sb a0, 4(a1)
.insn s STORE 1, a0, %lo12(my_symbol)(a1)
                               // equivalent to
                               // sh a0,%lo12(my_symbol)(a1)
```

The `.insn` directives can also be used in inline assembler:

```
int insn_example(int lhs, int rhs)
{
    int res;
    __asm(".insn r 0x33, 0x7, 0x0, %0, %1, %2" :
          "=r"(res) : "r"(lhs), "r"(rhs) );
          // generates AND r, r, r
    return res;
}
```

Function directives

Syntax

`CALL_GRAPH_ROOT` *`function`* `[,`*`category`*`]`

Parameters

function	The function, a symbol.
category	An optional call graph root category, a string.

Description

Use this directive to specify that, for stack usage analysis purposes, the function *`function`* is a call graph root. You can also specify an optional category, a quoted string.

The compiler will generate this directive in assembler list files, when needed.

Example

```
CALL_GRAPH_ROOT my_interrupt, "interrupt"
```

See also

Call frame information directives for stack usage analysis, page 127, for information about `CFI` directives required for stack usage analysis.

IAR C/C++ Development Guide for RISC-V for information about how to enable and use stack usage analysis.

Call frame information directives for names blocks

Syntax

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 BASEADDRESS` *`cfa type`* `[,` *`cfa type`*`] …`

Parameters

`bits`	The size of the resource in bits.
`cfa`	The name of a CFA (canonical frame address).
`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.
`size`	The size of the frame cell in bytes.
`type`	The segment memory type, such as `CODE`, `CONST`, or `DATA`. In addition, any of the memory types supported by the IAR ILINK Linker. It is only used for denoting an address space.

Description

Use these directives to define a names block:

Directive	Description
`CFI BASEADDRESS`	Declares a base address CFA (Canonical Frame Address).
`CFI ENDNAMES`	Ends a names block.
`CFI FRAMECELL`	Creates a reference into the caller's frame.
`CFI NAMES`	Starts a names block.
`CFI RESOURCE`	Declares a resource.

Table 26: Call frame information directives names block

Directive	Description
`CFI RESOURCEPARTS`	Declares a composite resource.
`CFI STACKFRAME`	Declares a stack frame CFA.
`CFI VIRTUALRESOURCE`	Declares a virtual resource.

Table 26: Call frame information directives names block (Continued)

Example

Examples of using CFI directives, page 36

See also

Tracking call frame usage, page 29

Call frame information directives for common blocks

Syntax

Common block directives:

```
CFI COMMON name USING namesblock
CFI ENDCOMMON name
CFI CODEALIGN codealignfactor
CFI DATAALIGN dataalignfactor
CFI DEFAULT { UNDEFINED | SAMEVALUE }
CFI RETURNADDRESS resource type
```

Parameters

codealignfactor	The smallest common 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 reduces the produced call frame information in size. The possible range is 1–256.
commonblock	The name of a previously defined common block.
dataalignfactor	The smallest common 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 reduces the produced call frame information in size. The possible ranges are –256 to –1 and 1 to 256.
name	The name of the block.
namesblock	The name of a previously defined names block.
resource	The name of a resource.

type	The memory type, such as `CODE`, `CONST`, or `DATA`. In addition, any of the segment memory types supported by the IAR ILINK Linker. It is only used for denoting an address space.

Description

Use these directives to define a common block:

Directive	Description
`CFI CODEALIGN`	Declares code alignment.
`CFI COMMON`	Starts or extends a common block.
`CFI DATAALIGN`	Declares data alignment.
`CFI DEFAULT`	Declares the default state of all resources.
`CFI ENDCOMMON`	Ends a common block.
`CFI RETURNADDRESS`	Declares a return address column.

Table 27: Call frame information directives common block

In addition to these directives you might also need the call frame information directives for specifying rules, or CFI expressions for resources and CFAs, see *Call frame information directives for tracking resources and CFAs*, page 124.

Example

Examples of using CFI directives, page 36

See also

Tracking call frame usage, page 29

Call frame information directives for data blocks

Syntax

```
CFI BLOCK name USING commonblock
CFI ENDBLOCK name
CFI { NOFUNCTION | FUNCTION label }
CFI { INVALID | VALID }
CFI { REMEMBERSTATE | RESTORESTATE }
CFI PICKER
CFI CONDITIONAL label [, label] …
```

Parameters

commonblock	The name of a previously defined common block.
label	A function label.

name	The name of the block.

Description

These directives allow call frame information to be defined in the assembler source code:

Directive	Description
`CFI BLOCK`	Starts a data block.
`CFI CONDITIONAL`	Declares a data block to be a conditional thread.
`CFI ENDBLOCK`	Ends a data block.
`CFI FUNCTION`	Declares a function associated with a data block.
`CFI INVALID`	Starts a range of invalid call frame information.
`CFI NOFUNCTION`	Declares a data block to not be associated with a function.
`CFI PICKER`	Declares a data block to be a picker thread. Used by the compiler for keeping track of execution paths when code is shared within or between functions.
`CFI REMEMBERSTATE`	Remembers the call frame information state.
`CFI RESTORESTATE`	Restores the saved call frame information state.
`CFI VALID`	Ends a range of invalid call frame information.

Table 28: Call frame information directives for data blocks

In addition to these directives, you might also need the call frame information directives for specifying rules, or CFI expressions for resources and CFAs, see *Call frame information directives for tracking resources and CFAs*, page 124.

Example

Examples of using CFI directives, page 36

See also

Tracking call frame usage, page 29

Call frame information directives for tracking resources and CFAs

Syntax

```
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

cfa	The name of a CFA (canonical frame address).
cfiexpr	A CFI expression, which can be one of these: ● A CFI operator with operands ● A numeric constant ● A CFA name ● A resource name.
constant	A constant value or an assembler expression that can be evaluated to a constant value.
offset	The offset relative the CFA. An integer with an optional sign.
resource	The name of a resource.

Unary operators

Overall syntax: `OPERATOR(operand)`

CFI 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 29: Unary operators in CFI expressions

Binary operators

Overall syntax: `OPERATOR(operand1,operand2)`

CFI operator	Operands	Description
`ADD`	*cfiexpr*,*cfiexpr*	Addition
`AND`	*cfiexpr*,*cfiexpr*	Bitwise `AND`
`DIV`	*cfiexpr*,*cfiexpr*	Division
`EQ`	*cfiexpr*,*cfiexpr*	Equal to
`GE`	*cfiexpr*,*cfiexpr*	Greater than or equal to
`GT`	*cfiexpr*,*cfiexpr*	Greater than
`LE`	*cfiexpr*,*cfiexpr*	Less than or equal to
`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.

Table 30: Binary operators in CFI expressions

CFI operator	Operands	Description
LT	*cfiexpr*,*cfiexpr*	Less than
MOD	*cfiexpr*,*cfiexpr*	Modulo
MUL	*cfiexpr*,*cfiexpr*	Multiplication
NE	*cfiexpr*,*cfiexpr*	Not equal to
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.
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 30: 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 that denotes a previously declared CFA. *size*, a constant expression that denotes a size in bytes. *offset*, a constant expression that denotes a size in bytes. Gets the value at address *cfa*+*offset* of size *size*.
IF	*cond*,*true*,*false*	Conditional operator. The operands are: *cond*, a CFI expression that denotes a condition. *true*, any CFI expression. *false*, any CFI 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 that denotes a size in bytes. *type*, a memory type. *addr*, a CFI expression that denotes a memory address. Gets the value at address *addr* in the segment memory type *type* of size *size*.

Table 31: Ternary operators in CFI expressions

Description

Use these directives to track resources and CFAs in common blocks and data blocks:

Directive	Description
`CFI cfa`	Declares the value of a CFA.
`CFI resource`	Declares the value of a resource.

Table 32: Call frame information directives for tracking resources and CFAs

Example

Examples of using CFI directives, page 36

See also

Tracking call frame usage, page 29

Call frame information directives for stack usage analysis

Syntax

```
CFI FUNCALL { caller } callee
CFI INDIRECTCALL { caller }
CFI NOCALLS { caller }
CFI TAILCALL { callee }
```

Parameters

callee	The label of the called function.
caller	The label of the calling function.

Description

These directives allow call frame information to be defined in the assembler source code:

Directive	Description
`CFI FUNCALL`	Declares function calls for stack usage analysis.
`CFI INDIRECTCALL`	Declares indirect calls for stack usage analysis.
`CFI NOCALLS`	Declares absence of calls for stack usage analysis.
`CFI TAILCALL`	Declares tail calls for stack usage analysis.

Table 33: Call frame information directives for stack usage analysis

See also

Tracking call frame usage, page 29

The *IAR C/C++ Development Guide for RISC-V* for information about stack usage analysis.

Pragma directives

This chapter describes the pragma directives of the IAR Assembler for RISC-V.

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 lists the pragma directives of the assembler:

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

Table 34: Pragma directives summary

Descriptions of pragma directives

The following pages describe each pragma directive.

Note that all pragma directives using = for value assignment should be entered like:

`#pragma` *`pragmaname`*`=`*`pragmavalue`*

or

`#pragma` *`pragmaname`* `=` *`pragmavalue`*

diag_default

Syntax

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

Parameters

`tag`	The number of a diagnostic message, for example the message number `Pe117`.

Description
Use this pragma directive to change the severity level back to the default, or to the severity level defined on the command line by any of the options `--diag_error`, `--diag_remark`, `--diag_suppress`, or `--diag_warning`, for the diagnostic messages specified with the tags.

Example
```
#pragma diag_default=Pe117
```

See also
The chapter *Diagnostics*.

diag_error

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

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description
Use this pragma directive to change the severity level to `error` for the specified diagnostic messages.

Example
```
#pragma diag_error=Pe117
```

See also
The chapter *Diagnostics*.

diag_remark

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

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description
Use this pragma directive to change the severity level to `remark` for the specified diagnostic messages.

Example
```
#pragma diag_remark=Pe177
```

See also
The chapter *Diagnostics*.

diag_suppress

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

Parameters

tag	The number of a diagnostic message, for example the message number `Pe117`.

Description Use this pragma directive to suppress the specified diagnostic messages.

Example `#pragma diag_suppress=Pe117,Pe177`

See also The chapter *Diagnostics*.

diag_warning

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

Parameters

tag	The number of a diagnostic message, for example the message number `Pe826`.

Description Use this pragma directive to change the severity level to `warning` for the specified diagnostic messages.

Example `#pragma diag_warning=Pe826`

See also The chapter *Diagnostics*.

message

Syntax `#pragma message(`*`string`*`)`

Parameters

string	The message that you want to direct to the standard output stream.

Description Use this pragma directive to make the assembler print a message on `stdout` when the file is assembled.

Example

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

Diagnostics

The following pages describe 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 IAR Embedded Workbench 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 58.

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 55.

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.

For information about the assembler options that are available for setting severity levels, see *Summary of assembler options*, page 41.

For information about the pragma directives that are available for setting severity levels, see the chapter *Pragma directives*.

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

O

P

R

S

T

U

V

W

X

Symbols