

R32C IAR Assembler

Reference Guide

for the Renesas

R32C/I00 Microcomputer Family

COPYRIGHT NOTICE

© Copyright 2007 IAR Systems. All rights reserved.

No part of this document may be reproduced without the prior written consent of IAR Systems. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

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, IAR Systems, IAR Embedded Workbench, IAR MakeApp, C-SPY, visualSTATE, From Idea To Target, IAR KickStart Kit and IAR PowerPac are trademarks or registered trademarks owned by IAR Systems AB.

Renesas is a registered trademark of Renesas Technology Corporation. R32C/100 is a registered trademark of Renesas Technology Corporation.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

EDITION NOTICE

First edition: April 2007

Part number: AR32C-1

This guide applies to version 1.x of the R32C IAR Embedded Workbench® IDE.

Internal reference: AFE2, T7:7c, IJOA

Contents

Tables	vii
Preface	ix
Who should read this guide	ix
How to use this guide	ix
What this guide contains	x
Other documentation	x
Document conventions	xi
Introduction to the R32C IAR Assembler	1
Introduction to assembler programming	1
Getting started	1
Modular programming	2
External interface details	2
Assembler invocation syntax	2
Passing options	3
Environment variables	4
Error return codes	4
Source format	5
Assembler instructions	5
Expressions, operands, and operators	6
Integer constants	6
ASCII character constants	7
Floating-point constants	7
TRUE and FALSE	8
Symbols	8
Labels	8
Register symbols	9
Predefined symbols	10
Absolute and relocatable expressions	12
Expression restrictions	12

List file format	13
Header	13
Body	13
Summary	14
Symbol and cross-reference table	14
Programming hints	14
Accessing special function registers	14
Using C-style preprocessor directives	14
Specifying immediate operands	15
Assembler options	17
Setting assembler options	17
Specifying parameters	18
Summary of assembler options	19
Description of assembler options	20
Assembler operators	33
Precedence of operators	33
Summary of assembler operators	34
Parenthesis operator – 1	34
Function operators – 2	34
Unary operators – 3	34
Multiplicative arithmetic operators – 4	35
Additive arithmetic operators – 5	35
Shift operators – 6	35
Comparison operators – 7	35
Equivalence operators – 8	35
Logical operators – 9-14	35
Conditional operator – 15	36
Description of assembler operators	36
Assembler directives	49
Summary of assembler directives	49
Module control directives	52
Syntax	53

Parameters	53
Descriptions	54
Symbol control directives	56
Syntax	56
Parameters	56
Descriptions	57
Examples	58
Segment control directives	58
Syntax	59
Parameters	59
Descriptions	60
Examples	62
Value assignment directives	63
Syntax	63
Parameters	64
Descriptions	64
Examples	65
Conditional assembly directives	66
Syntax	66
Parameters	67
Descriptions	67
Examples	67
Macro processing directives	68
Syntax	68
Parameters	69
Descriptions	69
Examples	72
Listing control directives	73
Syntax	74
Descriptions	74
Examples	75
C-style preprocessor directives	76
Syntax	77
Parameters	77

Descriptions	77
Examples	80
Data definition or allocation directives	81
Syntax	82
Parameters	82
Descriptions	82
Examples	83
Assembler control directives	83
Syntax	84
Parameters	84
Descriptions	84
Examples	85
Call frame information directives	86
Syntax	87
Parameters	88
Descriptions	89
Simple rules	93
CFI expressions	95
Example	97
Pragma directives	101
Summary of pragma directives	101
Descriptions of pragma directives	101
Diagnostics	103
Message format	103
Severity levels	103
Setting the severity level	104
Internal error	104
Index	105

Tables

1: Typographic conventions used in this guide	xi
2: Assembler environment variables	4
3: Assembler error return codes	4
4: Integer constant formats	6
5: ASCII character constant formats	7
6: Floating-point constants	7
7: Predefined register symbols	9
8: Predefined symbols	10
9: Symbol and cross-reference table	14
10: Assembler options summary	19
11: Generating a list of dependencies (--dependencies)	22
12: Conditional list options (-l)	27
13: Directing preprocessor output to file (--preprocess)	31
14: Assembler directives summary	49
15: Module control directives	53
16: Symbol control directives	56
17: Segment control directives	58
18: Value assignment directives	63
19: Conditional assembly directives	66
20: Macro processing directives	68
21: Listing control directives	73
22: C-style preprocessor directives	76
23: Data definition or allocation directives	81
24: Using data definition or allocation directives	82
25: Assembler control directives	83
26: Call frame information directives	86
27: Unary operators in CFI expressions	95
28: Binary operators in CFI expressions	95
29: Ternary operators in CFI expressions	96
30: Code sample with backtrace rows and columns	97
31: Pragma directives summary	101

Preface

Welcome to the R32C IAR Assembler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the R32C IAR Assembler to develop your application according to your requirements.

Who should read this guide

You should read this guide if you plan to develop an application, or part of an application, using assembler language for the R32C/100 microcomputer and need to get detailed reference information on how to use the R32C IAR Assembler. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the R32C/100 microcomputer. Refer to the documentation from Renesas for information about the R32C/100 microcomputer
- 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 R32C IAR Assembler, you should read the chapter *Introduction to the R32C IAR Assembler* in this reference guide.

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

If you are new to using the IAR Systems toolkit, we recommend that you first read the initial chapters of the *IAR Embedded Workbench® IDE User Guide*. They give product overviews, as well as tutorials that can help you get started. The *IAR Embedded Workbench® IDE User Guide* also contains a glossary.

What this guide contains

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

- *Introduction to the R32C IAR Assembler* provides programming information. It also describes the source code format, and the format of assembler listings.
- *Assembler options* first explains how to set the assembler options from the command line and how to use environment variables. It then gives an alphabetical summary of the assembler options, and contains detailed reference information about each option.
- *Assembler operators* gives a summary of the assembler operators, arranged in order of precedence, and provides detailed reference information about each operator.
- *Assembler directives* gives an alphabetical summary of the assembler directives, and provides detailed reference information about each of the directives, classified into groups according to their function.
- *Pragma directives* describes the pragma directives available in the assembler.
- *Diagnostics* contains information about the formats and severity levels of diagnostic messages.

Other documentation

The complete set of IAR Systems development tools for the R32C/100 microcomputer is described in a series of guides and online help files. For information about:

- Using the IAR Embedded Workbench® IDE including the IAR C-SPY® Debugger, refer to the *IAR Embedded Workbench® IDE User Guide*
- Programming for the R32C IAR C/C++ Compiler, refer to the *R32C IAR C/C++ Compiler Reference Guide*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library, refer to the online help system.

All of these guides are delivered in hypertext PDF or HTML format on the installation media. Some of them are also delivered as printed books.

Document conventions

This guide uses the following typographic conventions:



Style	Used for
computer	Text that you enter or that appears on the screen.
parameter	A label representing the actual value you should enter as part of a command.
[option]	An optional part of a command.
{option}	An mandatory part of a command.
a b c	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	A cross-reference within this guide or to another guide.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench interface.
	Identifies instructions specific to the command line interface.

Table 1: Typographic conventions used in this guide

Introduction to the R32C IAR Assembler

This chapter contains the following sections:

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

Introduction to assembler programming

Even if you do not intend to write a complete application in assembler language, there may be situations where you will find it necessary to write parts of the code in assembler, for example, when using mechanisms in the R32C/100 microcomputer that require precise timing and special instruction sequences.

To write efficient assembler applications, you should be familiar with the architecture and instruction set of the R32C/100 microcomputer. Refer to the Renesas hardware documentation for syntax descriptions of the instruction mnemonics.

GETTING STARTED

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

- Work through the tutorials—especially the one about mixing C and assembler modules—that you find in the *IAR Embedded Workbench® IDE User Guide*
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the *R32C IAR C/C++ Compiler Reference Guide*

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

Modular programming

Typically, you write your assembler code in assembler source files. In each source file, you define one or several assembler *modules* by using the module control directives. By structuring your code in small modules—in contrast to one single monolithic module—you can organize your application code in a logical structure, which makes the code easier to understand, and which benefits:

- an efficient program development
- reuse of modules
- maintenance.

Each module has a name and a type, where the type can be either `PROGRAM` or `LIBRARY`. The linker will always include a `PROGRAM` module, whereas a `LIBRARY` module is only included in the linked code if other modules reference a public symbol in the module. A module consists of one or more segments.

A *segment* is a logical entity containing a piece of data or code that should be mapped to a physical location in memory. You place your code and data in segments by using the segment control directives. A segment can be either *absolute* or *relocatable*. An absolute segment always has a fixed address in memory, whereas the address for a relocatable segment is resolved at link time. By using segments, you can control how your code and data will be placed in memory. Each segment consists of many *segment parts*. A segment part is the smallest linkable unit, which allows the linker to include only those units that are referred to.

External interface details

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

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

ASSEMBLER INVOCATION SYNTAX

The invocation syntax for the assembler is:

```
ar32c [options] [sourcefile] [options]
```

For example, when compiling the source file `prog.c`, use the following command to generate an object file with debug information:

```
ar32c prog --debug
```

The R32C IAR Assembler uses the default filename extensions `s53`, `asm`, and `msa` for source files.

Generally, the order of options on the command line, both relative to each other and to the source filename, is *not* significant. There is, however, one exception: when you use the `-I` option, the directories are searched in the same order that they are specified on the command line. The default filename extension for assembler output is `r53`.

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

There are three different ways of passing options to the assembler:

- Directly from the command line
Specify the options on the command line after the `ar32c` command; see *Assembler invocation syntax*, page 2.
- Via environment variables
The assembler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 4.
- Via a text file by using the `-f` option; see *-f*, page 26.

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

ENVIRONMENT VARIABLES

Assembler options can also be specified in the `ASMR32C` environment variable. The assembler automatically appends the value of this variable to every command line, so it provides a convenient method of specifying options that are required for every assembly.

The following environment variables can be used with the R32C IAR Assembler:

Environment variable	Description
ASMR32C	Specifies command line options; for example: set ASMR32C=-L -ws
AR32C_INC	Specifies directories to search for include files; for example: set AR32C_INC=c:\myinc\

Table 2: Assembler environment variables

For example, setting the following environment variable will always generate a list file with the name `temp.lst`:

```
ASMR32C=-l temp.lst
```

For information about the environment variables used by the IAR XLINK Linker and the IAR XLIB Librarian, see the *IAR Linker and Library Tools Reference Guide*.

ERROR RETURN CODES

When using the R32C IAR Assembler from within a batch file, you may need to determine whether the assembly was successful in order to decide what step to take next. For this reason, the assembler returns the following error return codes:

Return code	Description
0	Assembly successful, warnings may appear.
1	There were warnings, provided that the option <code>--warnings_affect_exit_code</code> was used.
2	There were non-fatal errors or fatal assembly errors (making the assembler abort).
3	There were crashing errors.

Table 3: Assembler error return codes

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

<i>label</i>	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.
<i>operation</i>	An assembler instruction or directive. This must not start in the first column—there must be some whitespace to the left of it.
<i>operands</i>	An assembler instruction or directive can have zero, one, or more operands. The operands are separated by commas. An operand can be: <ul style="list-style-type: none"> • a constant representing a numeric value or an address • a symbolic name representing a numeric value or an address (where the latter also is referred to as a label) • a floating-point constant • a register • a predefined symbol • \$, the program location counter (PLC) • an expression.
<i>comment</i>	Comment, preceded by a ; (semicolon) C or C++ comments are also allowed.

The components are separated by spaces or tabs.

A source line may not exceed 2047 characters.

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

Assembler instructions

The R32C IAR Assembler supports the syntax for assembler instructions as described in the chip manufacturer's hardware documentation.

Expressions, operands, and operators

Expressions consist of expression operands and operators.

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

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see also *Precedence of operators*, page 33.

The following 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), \$.

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

INTEGER CONSTANTS

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

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

Commas and decimal points are not permitted.

The following types of number representation are supported:

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

Table 4: 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 may be used in ASCII strings. If the quote character itself is to be accessed, two consecutive quotes must be used:

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

Table 5: ASCII character constant formats

FLOATING-POINT CONSTANTS

The R32C IAR Assembler will accept floating-point values as constants and convert them into IEEE double-precision (signed 64-bit) floating-point format or fractional format.

Floating-point numbers can be written in the format:

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

The following table shows some valid examples:

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

Table 6: Floating-point constants

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

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

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

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

TRUE AND FALSE

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

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

SYMBOLS

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

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

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

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

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

Note that symbols and labels are byte addresses.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

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

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

```
JMP $ ; Loop forever
```

REGISTER SYMBOLS

The following table shows the existing predefined register symbols:

Name	Address size	Description
R0L-R3L, R0H-R3H	8 bits	General purpose data registers
R0-R7	16 bits	General purpose data registers
R2R0, R3R1, R6R4, R7R5	32 bits	General purpose data register pairs
R3R1R2R0, R7R5R6R4	64 bits	General purpose data register combinations
A0-A3	32 bits	General purpose address registers
A1A0, A3A2	64 bits	General purpose address register combinations
SB	32 bits	Static base register
FB	32 bits	Frame base register
R0LB-R3LB, R0HB-R3HB	8 bits	General purpose data registers, register bank B
R0B-R7B	16 bits	General purpose data registers, register bank B
R2R0B, R3R1B, R6R4B, R7R5B	32 bits	General purpose data register pairs, register bank B
R3R1R2R0B, R7R5R6R4B	64 bits	General purpose data register combinations, register bank B
A0B-A3B	32 bits	General purpose address registers, register bank B
A1A0B, A3A2B	64 bits	General purpose address register combinations, register bank B
INTB	32 bits	The interrupt table base pointer
SVF	32 bits	Save flag register
SVP	32 bits	Save PC register
FLG	32 bits	Flag register
SP	32 bits	The active stack pointer (USB or ISB depending on the FLG register U bit)
ISP	32 bits	Interrupt stack pointer
VCT	32 bits	Vector register

Table 7: Predefined register symbols

Name	Address size	Description
DSA0–DSA3	32 bits	DMA source address registers
DSR0–DSR3	32 bits	DMA source address reload registers
DDA0–DDA3	32 bits	DMA destination address registers
DDR0–DDR3	32 bits	DMA destination address reload registers
DMD0–DMD3	32 bits	DMA mode registers
DCT0–DCT3	24 bits	DMA terminal count registers
DCR0–DCR3	24 bits	DMA terminal count reload registers
C	1 bit	Carry flag
D	1 bit	Debug flag
Z	1 bit	Zero flag
S	1 bit	Sign flag
B	1 bit	Register bank select flag
O	1 bit	Overflow flag
I	1 bit	Interrupt enable flag
U	1 bit	Stack pointer select flag

Table 7: Predefined register symbols (Continued)

PREDEFINED SYMBOLS

The R32C IAR Assembler defines a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in preprocessor directives or include them in the assembled code. The strings returned by the assembler are enclosed in double quotes.

The following predefined symbols are available:

Symbol	Value
__AR32C__	An integer that is set to 1 when the code is assembled with the R32C IAR Assembler.
__BUILD_NUMBER__	A unique integer that identifies the build number of the assembler currently in use. The build number does not necessarily increase with an assembler that is released later.
__CODE_MODEL__	An integer that identifies the code model in use. The symbol reflects the <code>--code_model</code> option and is defined to <code>__FAR__</code> or <code>__HUGE__</code> . These symbolic names can be used when testing the <code>__CODE_MODEL__</code> symbol.

Table 8: Predefined symbols

Symbol	Value
__DATA_MODEL__	An integer that identifies the data model in use. The symbol reflects the <code>--data_model</code> option and can be defined to <code>__NEAR__</code> , <code>__FAR__</code> , or <code>__HUGE__</code> .
__DATE__	The current date in <code>dd/Mmm/yyyy</code> format (string).
__DOUBLE__	Either 32 or 64, depending on the setting of the option <code>--double</code> .
__FILE__	The name of the current source file (string).
__IAR_SYSTEMS_ASM__	IAR assembler identifier (number).
__LINE__	The current source line number (number).
__SUBVERSION__	An integer that identifies the version letter of the version number, for example the C in 4.21C, as an ASCII character.
__TIME__	The current time in <code>hh:mm:ss</code> format (string).
__VER__	The version number in integer format; for example, version 4.17 is returned as 417 (number).

Table 8: Predefined symbols (Continued)

Including symbol values in code

There are several data definition directives provided to 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:

```
tim      DC8      __TIME__      ; Time string
          ...
          MOVA     tim,A0        ; Load time string in A0
          JSR      printstr      ; Call string output routine
```

Testing symbols for conditional assembly

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

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

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

```

```
#endif
```

See *Conditional assembly directives*, page 66.

ABSOLUTE AND RELOCATABLE EXPRESSIONS

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

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

Such expressions are evaluated and resolved at link time, by the IAR XLINK Linker. There are no restrictions on the expression; any operator can be used on symbols from any segment, or any combination of segments.

For example, a program could define the segments DATA and CODE as follows:

```

NAME      prog1
EXTERN    third
RSEG      DATA
first:    DC8      5
second:   DC8      3
ENDMOD
MODULE    prog2
RSEG      CODE
start     ...
```

Then in the segment CODE, the following relocatable expressions are legal:

```

DC8      first
DC8      first+1
DC8      1+first
DC8      (first/second)*third
```

Note: At assembly time, there will be no range check. The range check will occur at link time and, if the values are too large, there will be 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 (segment 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 may 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 will, 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 segment. 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 will be 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 has been included in the source file, a symbol and cross-reference table is produced.

The following information is provided for each symbol in the table:

Information	Description
Symbol	The symbol's user-defined name.
Mode	ABS (Absolute), or REL (Relocatable).
Segments	The name of the segment 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 segment part.

Table 9: Symbol and cross-reference table

Programming hints

This section gives hints on how to write efficient code for the R32C IAR Assembler. For information about projects including both assembler and C or C++ source files, see the *R32C IAR C/C++ Compiler Reference Guide*.

ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of R32C/100 derivatives are included in the IAR Systems product package, in the `\r32c\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 R32C IAR C/C++ Compiler, and they are suitable to use as templates when creating new header files for other R32C/100 derivatives.

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

```
#ifdef __IAR_SYSTEMS_ASM__
    (assembler-specific defines)
#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 83.

SPECIFYING IMMEDIATE OPERANDS

The R32C/100 instruction set allows immediate operands to be encoded in a smaller format known as `#immex`. To use this format, it is best to use negative numbers rather than trying to sign-extend them in code. A sign-extended constant such as `#0xFFFFE024` will be interpreted as the unsigned entity rather than a negative value, and will cause a range error. To use the `#immex` format, instead write `#-0x1FDC`."

Assembler options

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



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

Setting assembler options

To set assembler options from the command line, include them on the command line after the `ar32c` command, either before or after the source filename. For example, when assembling the source `prog.s53`, use the following command to generate an object file with debug information:

```
ar32c prog --debug
```

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

```
ar32c prog -l prog.lst
```

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

```
ar32c prog -DDEBUG=1
```

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

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

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

SPECIFYING PARAMETERS

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

For instance, an include file path of `\usr\include` can be specified either as:

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

or as

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

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

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

When a parameter is needed for an option with a long name, it can be specified 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 may be repeated, and may also have comma-separated values (without space), for example:

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

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

```
ar32c 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 (`--`). The following example will generate a list on standard output:

```
ar32c prog -l ---
```

Summary of assembler options

The following table summarizes the assembler options available from the command line:

Command line option	Description
--case_insensitive	Case-insensitive user symbols
--code_model	Defines the symbol <code>__CODE_MODEL__</code>
-D	Defines preprocessor symbols
--data_model	Defines the symbol <code>__DATA_MODEL__</code>
--debug	Generates debug information
--dependencies	Lists file dependencies
--diag_error	Treats these diagnostics as errors
--diag_remark	Treats these diagnostics as remarks
--diag_suppress	Suppresses these diagnostics
--diag_warning,	Treats these diagnostics as warnings
--diagnostics_tables	Lists all diagnostic messages
--dir_first	Allows directives in the first column
--double	Defines the symbol <code>__DOUBLE__</code>
--enable_multibytes	Enables support for multibyte characters
--error_limit	Specifies the allowed number of errors before the assembler stops
-f	Extends the command line
--header_context	Lists all referred source files
-I	Includes file paths
-l	Output list file
-M	Macro quote characters
--mnem_first	Allows mnemonics in the first column
--no_path_in_file_macros	Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code>
--no_warnings	Disables all warnings
--no_wrap_diagnostics	Disables wrapping of diagnostic messages
-o	Sets object filename
--only_stdout	Uses standard output only
--preinclude	Includes an include file before reading the source file

Table 10: Assembler options summary

Command line option	Description
--preprocess	Preprocessor output to file
-r	Generates debug information
--remarks	Enables remarks
--silent	Sets silent operation
--warnings_affect_exit_code	Warnings affect exit code
--warnings_are_errors	Treats all warnings as errors

Table 10: Assembler options summary (Continued)

Description of assembler options

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



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

--case_insensitive --case_insensitive

Use this option to make user symbols case insensitive.

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

You can also use the assembler directives CASEON and CASEOFF to control case sensitivity for user-defined symbols. See *Assembler control directives*, page 83, for more information.

Note: The --case_insensitive option does not affect preprocessor symbols. Preprocessor symbols are always case sensitive, regardless of whether they are defined in the IAR Embedded Workbench IDE or on the command line. See *Defining and undefining preprocessor symbols*, page 78.



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

```
--code_model --code_model={ far | f | huge | h }
```

Parameters

far (default)	Sets the predefined symbol <code>__CODE_MODEL__</code> to <code>__CODE_MODEL_FAR__</code>
huge	Sets the predefined symbol <code>__CODE_MODEL__</code> to <code>__CODE_MODEL_HUGE__</code>

Description

Use this option to define the symbol `__CODE_MODEL__`. See *Predefined symbols*, page 10.



Project>Options>General Options>Target>Code model

```
-D -Dsymbol [=value]
```

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

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

Example

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

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

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

```
Production version: ar32c prog
Test version:      ar32c prog -DTESTVER
```

Alternatively, your source might use a variable that you need to 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:

```
ar32c prog -DFRAME_RATE=3
```



Project>Options>Assembler>Preprocessor>Defined symbols

--data_model --data_model={near|n|far|f|huge|h}

Parameters

near	Sets the predefined symbol <code>__DATA_MODEL__</code> to <code>__DATA_MODEL_NEAR__</code>
far (default)	Sets the predefined symbol <code>__DATA_MODEL__</code> to <code>__DATA_MODEL_FAR__</code>
huge	Sets the predefined symbol <code>__DATA_MODEL__</code> to <code>__DATA_MODEL_HUGE__</code>

Description

Use this option to define the symbol `__DATA_MODEL__`. See *Predefined symbols*, page 10.



Project>Options>General Options>Target>Data model

--debug, -r --debug
-r

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

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



Project>Options>Assembler >Output>Generate debug information

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

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

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

Table 11: Generating a list of dependencies (`--dependencies`)

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

If a *directory* is specified, the assembler stores the output in that directory, in a file with the extension *i*. The filename will be the same as the name of the assembled source

file, unless a different name has been specified with the option `-o`, in which case that name will be used.

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

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

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

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

```
foo.r53: c:\iar\product\include\stdio.h
foo.r53: d:\myproject\include\foo.h
```

Example 1

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

```
ar32c prog --dependencies=i listing
```

Example 2

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

```
ar32c prog --dependencies \mypath\listing
```

Note: Both `\` and `/` can be used as directory delimiters.

Example 3

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

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

```
%.r53 : %.c
$(ASM) $(ASMFLAGS) $< --dependencies=m $*.d
```

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

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

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

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



This option is not available in the IAR Embedded Workbench IDE.

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

Use this option to classify diagnostic messages as errors.

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

The following example classifies warning As001 as an error:

```
--diag_error=As001
```



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

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

Use this option to classify diagnostic messages as remarks.

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

The following example classifies the warning As001 as a remark:

```
--diag_remark=As001
```



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

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

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

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



Project>Options>Assembler >Diagnostics>Suppress these diagnostics

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

Use this option to classify diagnostic messages as warnings.

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

The following example classifies the remark As028 as a warning:

```
--diag_warning=As028
```



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

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

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

This option cannot be given together with other options.

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

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

Example 1

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

```
--diagnostics_tables diag
```

Example 2

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

```
--diagnostics_tables .
```

Both `\` and `/` can be used as directory delimiters.



This option is not available in the IAR Embedded Workbench IDE.

`--dir_first` `--dir_first`

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

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



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

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

Parameters

- | | |
|---------------|--|
| far (default) | Sets the predefined symbol <code>__DOUBLE__</code> to 32 |
| huge | Sets the predefined symbol <code>__DOUBLE__</code> to 64 |

Description

Use this option to define the symbol `__DOUBLE__`. See *Predefined symbols*, page 10.



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

`--enable_multibytes` `--enable_multibytes`

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

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



Project>Options>Assembler>Language>Enable multibyte support

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

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



This option is not available in the IAR Embedded Workbench IDE.

`-f` `-f filename`

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

The `-f` option is particularly useful where there is a large number of options which are more conveniently placed in a file than on the command line itself. For example, to run the assembler with further options taken from the file `extend.xcl`, use:

```
ar32c prog -f extend.xcl
```



To set this option, use:

Project>Options>Assembler>Extra Options

`--header_context` `--header_context`

Occasionally, it is necessary to 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 IAR Embedded Workbench IDE.

-I *-Iprefix*

Adds the `#include` file search prefix *prefix*.

By default, the assembler searches for `#include` files only in the current working directory and in the paths specified in the `AR32C_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, will 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 `AR32C_INC` environment variable, provided that this variable is set.



Project>Options>Assembler >Preprocessor>Additional include directories

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

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

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

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

Table 12: Conditional list options (-l)

Command line option	Description
-lN	Do not include diagnostics

Table 12: Conditional list options (-l) (Continued)

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

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

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

Example 1

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

```
ar32c sourcefile -l list
```

Example 2

If you assemble the file *mysource.s53* and want to generate a listing to a file *mysource.lst* in the working directory, you could use:

```
ar32c mysource -l .
```

Note: Both ** and */* can be used as directory delimiters.



To set related options, select:

Project>Options>Assembler>List

-M -Mab

Specifies quote characters for macro arguments by setting the characters used for the left and right quotes of each macro argument to *a* and *b* respectively.

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

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

```
ar32c filename -M'<>'
```

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.



Project>Options>Assembler >Language>Macro quote characters

--mnem_first --mnem_first

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

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



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

--no_path_in_file_macros --no_path_in_file_macros

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



This option is not available in the IAR Embedded Workbench IDE.

--no_warnings --no_warnings

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



This option is not available in the IAR Embedded Workbench IDE.

--no_wrap_diagnostics --no_wrap_diagnostics

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



This option is not available in the IAR Embedded Workbench IDE.

-o -o {filename|directory}

Use the `-o` option to specify an output file.

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

If a *directory* is specified, the assembler stores the object code in that directory, in a file with the same name as the name of the assembled source file, but with the extension *r53*. To specify the working directory, replace *directory* with a period (*.*).

Example 1

To store the assembler output in a file called *obj.r53* in the *mypath* directory, you would use:

```
ar32c sourcefile -o \mypath\obj
```

Example 2

If you assemble the file *mysource.s53* and want to store the assembler output in a file *mysource.r53* in the working directory, you could use:

```
ar32c mysource -o .
```

Note: Both \ and / can be used as directory delimiters. You must include a space between the option itself and the filename.



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

--only_stdout --only_stdout

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



This option is not available in the IAR Embedded Workbench IDE.

--preinclude --preinclude *includefile*

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



To set this option, use:

Project>Options>Assembler>Extra Options

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

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

The following table shows the mapping of the available preprocessor modifiers:

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

Table 13: Directing preprocessor output to file (--preprocess)

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

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

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

Example 1

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

```
ar32c sourcefile --preprocess=c output
```

Example 2

If you assemble the file `mysource.s53` and want to store the assembler output with #line directives to a file `mysource.i` in the working directory, you could use:

```
ar32c mysource --preprocess=l .
```

Note: Both \ and / can be used as directory delimiters.



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

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

```
-r
```

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

In order 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

```
--remarks --remarks
```

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

See *Severity levels*, page 103, for additional information about diagnostic messages.



Project>Options>Assembler >Diagnostics>Enable remarks

```
--silent --silent
```

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

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



This option is not available in the IAR Embedded Workbench IDE.

```
--warnings_affect_exit_code --warnings_affect_exit_code
```

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



This option is not available in the IAR Embedded Workbench IDE.

```
--warnings_are_errors --warnings_are_errors
```

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

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

```
--diag_warning=As001
```

For additional information, see `--diag_warning`, page 24.



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

Assembler operators

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

Precedence of operators

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

The following 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, the following expression evaluates to 1:

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

Note: The precedence order in the R32C IAR Assembler closely follows the precedence order of the ANSI C++ standard for operators, where applicable.

Summary of assembler operators

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

PARENTHESIS OPERATOR – 1

()	Parenthesis.
-----	--------------

FUNCTION OPERATORS – 2

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

UNARY OPERATORS – 3

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

MULTIPLICATIVE ARITHMETIC OPERATORS – 4

<code>*</code>	Multiplication.
<code>/</code>	Division.
<code>MOD [%]</code>	Modulo.

ADDITIVE ARITHMETIC OPERATORS – 5

<code>+</code>	Addition.
<code>-</code>	Subtraction.

SHIFT OPERATORS – 6

<code>SHL [<<]</code>	Logical shift left.
<code>SHR [>>]</code>	Logical shift right.

COMPARISON OPERATORS – 7

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

EQUIVALENCE OPERATORS – 8

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

LOGICAL OPERATORS – 9-14

<code>BINAND [&]</code>	Bitwise AND (9).
<code>BINXOR [^]</code>	Bitwise exclusive OR (10).
<code>BINOR []</code>	Bitwise OR (11).

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

CONDITIONAL OPERATOR – 15

? :	Conditional operator.
-----	-----------------------

Description of assembler operators

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

-
- () Parenthesis (1).
- (and) group expressions to be evaluated separately, overriding the default precedence order.

Example

1+2*3 → 7
(1+2)*3 → 9

-
- * Multiplication (4).
- * produces the product of its two operands. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

Example

2*2 → 4
-2*2 → -4

-
- + Unary plus (3).
- Unary plus operator.

Example

+3 → 3
3*+2 → 6

+ Addition (5).

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

Example

92+19 → 111
-2+2 → 0
-2+-2 → -4

- Unary minus (3).

The unary minus operator performs arithmetic negation on its operand.

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

Example

-3 → -3
3*-2 → -6
4--5 → 9

- Subtraction (5).

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

Example

92-19 → 73
-2-2 → -4
-2--2 → 0

/ Division (4).

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

Example

9/2 → 4
-12/3 → -4
9/2*6 → 24

?: Conditional operator (15).

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

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

Syntax

condition ? *expr* : *expr*

Example

```
5 ? 6 : 7 → 6
0 ? 6 : 7 → 7
```

AND [&&] Logical AND (12).

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

Example

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

BINAND [&] Bitwise AND (9).

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

Example

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

BINNOT [~] Bitwise NOT (3).

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

BYTE3 Third byte (2).

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

Example

BYTE3 0x12345678 → 0x34

BYTE4 Fourth byte (2).

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

Example

BYTE4 0x12345678 → 0x12

DATE Current date/time (2).

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

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

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

Example

To assemble the date of assembly:

today: DC8 DATE 5, DATE 4, DATE 3

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

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

Example

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

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

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

Example

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

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

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

Example

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

HIGH High byte (2).

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

Example

```
HIGH 0xABCD → 0xAB
```

HWRD High word (2).

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

Example

```
HWRD 0x12345678 → 0x1234
```

LE [\leq] Less than or equal (7).

\leq evaluates to 1 (true) if the left operand has a lower or equal numeric value to the right operand, otherwise it will be 0 (false).

Example

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

LOW Low byte (2).

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

Example

```
LOW 0xABCD → 0xCD
```

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

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

Example

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

LWRD Low word (2).

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

Example

```
LWRD 0x12345678 → 0x5678
```

MOD [%] Modulo (4).

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

$X \text{ MOD } Y$ is equivalent to $X - Y * (X/Y)$ using integer division.

Example

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

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

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

Example

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

NOT [!] Logical NOT (3).

Use NOT to negate a logical argument.

Example

```
NOT 0101B → 0
NOT 0000B → 1
```

OR [| |] Logical OR (14).

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

Example

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

SFB Segment begin (2).

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

Syntax

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

Parameters

<i>segment</i>	The name of a relocatable segment, which must be defined before <i>SFB</i> is used.
<i>offset</i>	An optional offset from the start address. The parentheses are optional if <i>offset</i> is omitted.

Example

```

NAME    demo
RSEG    segtab:CONST
start:  DC16  SFB(mycode)
```

Even if the above code is linked with many other modules, *start* will still be set to the address of the first byte of the segment.

SFE Segment end (2).

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

Syntax

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

Parameters

<i>segment</i>	The name of a relocatable segment, which must be defined before <i>SFE</i> is used.
<i>offset</i>	An optional offset from the start address. The parentheses are optional if <i>offset</i> is omitted.

Example

```

        NAME    demo
        RSEG    segtab:CONST
end:    DC16    SFE(mycode)

```

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

The size of the segment `MY_SEGMENT` can be calculated as:

```
SFE(MY_SEGMENT) - SFB(MY_SEGMENT)
```

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

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

Example

```

00011100B SHL 3 → 11100000B
000001111111111111B SHL 5 → 11111111111100000B
14 SHL 1 → 28

```

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

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

Example

```

01110000B SHR 3 → 00001110B
111111111111111111B SHR 20 → 0
14 SHR 1 → 7

```

SIZEOF Segment size (2).

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

Syntax

```
SIZEOF (segment)
```

Parameters

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

Example

The following code sets `size` to the size of the segment `mycode`.

```
MODULE table
RSEG mycode:CODE ;forward declaration of mycode
RSEG segtab:CONST
size: DC32 SIZEOF(mycode)
ENDMOD

MODULE application
RSEG mycode:CODE
NOP ;placeholder for application code
ENDMOD
```

UGT Unsigned greater than (7).

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

Example

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

ULT Unsigned less than (7).

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

Example

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

UPPER Third byte (2).

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

Example

UPPER 0x12345678 → 0x34

XOR Logical exclusive OR (13).

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

Example

0101B XOR 1010B → 0

0101B XOR 0000B → 1

Assembler directives

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

Summary of assembler directives

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

- *Module control directives*, page 52
- *Symbol control directives*, page 56
- *Segment control directives*, page 58
- *Value assignment directives*, page 63
- *Conditional assembly directives*, page 66
- *Macro processing directives*, page 68
- *Listing control directives*, page 73
- *C-style preprocessor directives*, page 76
- *Data definition or allocation directives*, page 81
- *Assembler control directives*, page 83
- *Call frame information directives*, page 86.

The following table gives a summary of all the assembler directives.

Directive	Description	Section
<code>#define</code>	Assigns a value to a label.	C-style preprocessor
<code>#elif</code>	Introduces a new condition in a <code>#if...#endif</code> block.	C-style preprocessor
<code>#else</code>	Assembles instructions if a condition is false.	C-style preprocessor
<code>#endif</code>	Ends a <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> block.	C-style preprocessor
<code>#error</code>	Generates an error.	C-style preprocessor
<code>#if</code>	Assembles instructions if a condition is true.	C-style preprocessor
<code>#ifdef</code>	Assembles instructions if a symbol is defined.	C-style preprocessor
<code>#ifndef</code>	Assembles instructions if a symbol is undefined.	C-style preprocessor
<code>#include</code>	Includes a file.	C-style preprocessor

Table 14: Assembler directives summary

Directive	Description	Section
#line	Changes the line numbers.	C-style preprocessor
#pragma	Controls extension features.	C-style preprocessor
#undef	Undefines a label.	C-style preprocessor
/*comment*/	C-style comment delimiter.	Assembler control
//	C++ style comment delimiter.	Assembler control
=	Assigns a permanent value local to a module.	Value assignment
ALIGN	Aligns the program location counter by inserting zero-filled bytes.	Segment control
ALIGNRAM	Aligns the program location counter.	Segment control
ASEG	Begins an absolute segment.	Segment control
ASEGN	Begins a named absolute segment.	Segment control
ASSIGN	Assigns a temporary value.	Value assignment
BLOCK	Specifies the block number for an alias created by the SYMBOL directive.	Symbol control
CASEOFF	Disables case sensitivity.	Assembler control
CASEON	Enables case sensitivity.	Assembler control
CFI	Specifies call frame information.	Call frame information
COMMON	Begins a common segment.	Segment control
DC8	Generates 8-bit constants, including strings.	Data definition or allocation
DC16	Generates 16-bit constants.	Data definition or allocation
DC24	Generates 24-bit constants.	Data definition or allocation
DC32	Generates 32-bit constants.	Data definition or allocation
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

Table 14: Assembler directives summary (Continued)

Directive	Description	Section
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	Terminates 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
ENDMOD	Terminates the assembly of the current module.	Module control
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.	Segment control
EXITM	Exits prematurely from a macro.	Macro processing
EXTERN	Imports an external symbol.	Symbol control
IF	Assembles instructions if a condition is true.	Conditional assembly
IMPORT	Imports an external symbol.	Symbol control
LIBRARY	Begins a library module.	Module control
LIMIT	Checks a value against limits.	Value assignment
LOCAL	Creates symbols local to a macro.	Macro processing
LSTCND	Controls conditional assembly listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control

Table 14: Assembler directives summary (Continued)

Directive	Description	Section
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 library module.	Module control
NAME	Begins a program module.	Module control
ODD	Aligns the program location counter to an odd address.	Segment control
ORG	Sets the program location counter.	Segment control
OVERLAY	Recognized but ignored.	Symbol control
PROGRAM	Begins a program 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 relocatable segment.	Segment control
RTMODEL	Declares runtime model attributes.	Module control
SET	Assigns a temporary value.	Value assignment
SYMBOL	Creates an alias that can be used for referring to a C/C++ symbol.	Symbol control
VAR	Assigns a temporary value.	Value assignment

Table 14: Assembler directives summary (Continued)

Module control directives

Module control directives are used for marking the beginning and end of source program modules, and for assigning names and types to them. See *Expression restrictions*, page

12, for a description of the restrictions that apply when using a directive in an expression.

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

Table 15: Module control directives

SYNTAX

```

END [address]
ENDMOD [address]
LIBRARY symbol [(expr)]
MODULE symbol [(expr)]
NAME symbol [(expr)]
PROGRAM symbol [(expr)]
RTMODEL key, value

```

PARAMETERS

address An optional expression that determines the start address of the program. It can take any positive integer value.

expr An optional expression used by the compiler to encode the runtime options. It must be within the range 0-255 and evaluate to a constant value. The expression is only meaningful if you are assembling source code that originates as assembler output from the compiler.

key A text string specifying the key.

<i>symbol</i>	Name assigned to module, used by XLINK, XAR, and XLIB when processing object files.
<i>value</i>	A text string specifying the value.

DESCRIPTIONS

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 library module

Use `MODULE` or `LIBRARY` to create libraries containing a number of small modules—like runtime systems for high-level languages—where each module often represents a single routine. With the multi-module facility, you can significantly reduce the number of source and object files needed.

Library modules are only copied into the linked code if other modules reference a public symbol in the module.

Terminating a module

Use `ENDMOD` to define the end of a module.

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 terminates the last module in the file, if this is not done explicitly with an `ENDMOD` directive.

Assembling multi-module files

Program entries must be either relocatable or absolute, and will show up in XLINK load maps, as well as in some of the hexadecimal absolute output formats. Program entries must not be defined externally.

The following rules apply when assembling multi-module files:

- At the beginning of a new module all user symbols are deleted, except for those created by `DEFINE`, `#define`, or `MACRO`, the location counters are cleared, and the mode is set to absolute.

- Listing control directives remain in effect throughout the assembly.

Note: `END` must always be placed after the *last* module, and there must not be any source lines (except for comments and listing control directives) between an `ENDMOD` and the next module (beginning with `MODULE`, `LIBRARY`, `NAME`, or `PROGRAM`).

If any of the directives `NAME`, `MODULE`, `LIBRARY`, or `PROGRAM` is missing, the module will be assigned the name of the source file and the attribute `program`.

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 *R32C IAR C/C++ Compiler Reference Guide*.

Examples

The following example defines three modules where:

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

```
MODULE MOD_1
    RTMODEL    "foo", "1"
    RTMODEL    "bar", "XXX"
    ...
ENDMOD

MODULE MOD_2
    RTMODEL    "foo", "2"
    RTMODEL    "bar", "*"
    ...
ENDMOD
```

```
MODULE MOD_3
    RTMODEL    "bar", "XXX"
    ...
END
```

Symbol control directives

These directives control how symbols are shared between modules.

Directive	Description
BLOCK	Specifies the block number for an alias created by the SYMBOL directive.
EXTERN, IMPORT	Imports an external symbol.
OVERLAY	Recognized but ignored.
PUBLIC	Exports symbols to other modules.
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.
REQUIRE	Forces a symbol to be referenced.
SYMBOL	Creates an alias for a C/C++ symbol.

Table 16: Symbol control directives

SYNTAX

```
label BLOCK old_label, block_number
EXTERN symbol [,symbol] ...
IMPORT symbol [,symbol] ...
PUBLIC symbol [,symbol] ...
PUBWEAK symbol [,symbol] ...
REQUIRE symbol
label SYMBOL "C/C++_symbol" [,old_label]
```

PARAMETERS

<i>block_number</i>	Block number of the alias created by the SYMBOL directive.
<i>C/C++_symbol</i>	C/C++ symbol to create an alias for.
<i>label</i>	Label to be used as an alias for a C/C++ symbol.
<i>old_label</i>	Alias created earlier by a SYMBOL directive.
<i>symbol</i>	Symbol to be imported or exported.

DESCRIPTIONS

Exporting symbols to other modules

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

The `PUBLIC` directive always exports full 32-bit values, which makes it feasible to use global 32-bit constants also in assemblers for 8-bit and 16-bit processors. With the `LOW`, `HIGH`, `>>`, and `<<` operators, any part of such a constant can be loaded in an 8-bit or 16-bit register or word.

There are no restrictions on the number of `PUBLIC`-defined symbols in a module.

Exporting symbols with multiple definitions to other modules

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

A symbol defined as `PUBWEAK` must be a label in a segment part, and it must be the *only* symbol defined as `PUBLIC` or `PUBWEAK` in that segment part.

Note: Library modules are only linked if a reference to a symbol in that module is made, and that symbol has not already been 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 segment part containing the symbol must be loaded for the code containing the reference to work, but the dependence is not otherwise evident.

Referring to scoped C/C++ symbols

Use the `SYMBOL` directive to create an alias for a C/C++ symbol. The alias can be used for referring to the C/C++ symbol. The symbol and the alias must be located within the same scope.

Use the `BLOCK` directive to provide the block scope for the alias.

Typically, the `SYMBOL` and the `BLOCK` directives are for compiler internal use only, for example when referring to objects inside classes or namespaces. For detailed information about how to use these directives, declare and define your C/C++ symbol, compile, and view the assembler listfile output.

EXAMPLES

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

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

It defines `print` as an external routine; the address will be resolved at link time.

```
NAME      error
EXTERN    print
PUBLIC    err
RSEG      CODE24:CODE

str
    DC8    "*** Error ***"

err
    MOVA    str,A0
    JSR     print
    END
```

Segment control directives

The segment directives control how code and data are located. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
ALIGN	Aligns the program location counter by inserting zero-filled bytes.	No external references Absolute
ALIGNRAM	Aligns the program location counter.	No external references Absolute
ASEG	Begins an absolute segment.	No external references Absolute
ASEGN	Begins a named absolute segment.	No external references Absolute
COMMON	Begins a common segment.	No external references Absolute

Table 17: Segment control directives

Directive	Description	Expression restrictions
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
ORG	Sets the location counter.	No external references Absolute (see below)
RSEG	Begins a relocatable segment.	No external references Absolute

Table 17: Segment control directives (Continued)

SYNTAX

```

ALIGN align [, value]
ALIGNRAM align
ASEG [start]
ASEGN segment [:type], address
COMMON segment [:type] [(align)]
EVEN [value]
ODD [value]
ORG expr
RSEG segment [:type] [flag] [(align)]

```

PARAMETERS

<i>address</i>	Address where this segment part will be placed.
<i>align</i>	The power of two to which the address should be aligned, in most cases in the range 0 to 30. The default align value is 1, except for stack data where the default is 4.
<i>expr</i>	Address to set the location counter to.
<i>flag</i>	NOROOT, ROOT NOROOT means that the segment part is discarded by the linker if no symbols in this segment part are referred to. Normally, all segment parts except startup code and interrupt vectors should set this flag. The default mode is ROOT which indicates that the segment part must not be discarded.

REORDER, NOREORDER

REORDER allows the linker to reorder segment parts. For a given segment, all segment parts must specify the same state for this flag. The default mode is NOREORDER which indicates that the segment parts must remain in order.

SORT, NOSORT

SORT means that the linker will sort the segment parts in decreasing alignment order. For a given segment, all segment parts must specify the same state for this flag. The default mode is NOSORT which indicates that the segment parts will not be sorted.

<i>segment</i>	The name of the segment.
<i>start</i>	A start address that has the same effect as using an <code>ORG</code> directive at the beginning of the absolute segment.
<i>type</i>	The memory type, typically <code>CODE</code> or <code>DATA</code> . In addition, any of the types supported by the IAR XLINK Linker.
<i>value</i>	Byte value used for padding, default is zero.

DESCRIPTIONS

Beginning an absolute segment

Use `ASEG` to set the absolute mode of assembly, which is the default at the beginning of a module.

If the parameter is omitted, the start address of the first segment is 0, and subsequent segments continue after the last address of the previous segment.

Beginning a named absolute segment

Use `ASEGN` to start a named absolute segment located at the address *address*.

This directive has the advantage of allowing you to specify the memory type of the segment.

Beginning a relocatable segment

Use `RSEG` to set the current mode of the assembly to relocatable assembly mode. The assembler maintains separate location counters (initially set to zero) for all segments, which makes it possible to switch segments and mode anytime without the need to save the current segment location counter.

Up to 65536 unique, relocatable segments may be defined in a single module.

Beginning a common segment

Use `COMMON` to place data in memory at the same location as `COMMON` segments from other modules that have the same name. In other words, all `COMMON` segments of the same name will start at the same location in memory and overlay each other.

Obviously, the `COMMON` segment type should not be used for overlaid executable code. A typical application would be when you want a number of different routines to share a reusable, common area of memory for data.

It can be practical to have the interrupt vector table in a `COMMON` segment, thereby allowing access from several routines.

The final size of the `COMMON` segment is determined by the size of largest occurrence of this segment. The location in memory is determined by the `XLINK -Z` command; see the *IAR Linker and Library Tools Reference Guide*.

Use the `align` parameter in any of the above directives to align the segment start address.

Setting the program location counter (PLC)

Use `ORG` to set the program location counter of the current segment to the value of an expression. When `ORG` is used in an absolute segment (`ASEG`), the parameter expression must be absolute. However, when `ORG` is used in a relative segment (`RSEG`), the expression may be either absolute or relative (and the value is interpreted as an offset relative to the segment start in both cases).

The program location counter is set to zero at the beginning of an assembler module.

Aligning a segment

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

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

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

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

EXAMPLES

Beginning an absolute segment

The following example places a boot loader stub in the internal ROM area and jumps to main if the startup mode pin was high.

```

MODULE    boot_load
PUBLIC    boot_loader
EXTERN    main

        // locate the boot loader in the internal ROM memory area
ASEGN     CODE32:CODE:ROOT,0x60000

boot_loader:
    btst    1,0x3C0:16        ; test startup mode pin
    jz      ??start_load
    mova     main,a0
    jmpil.l  a0                ; pin was high, jump to main
??start_load:                  ; pin was low, start loading

```

Beginning a relocatable segment

In the following example, the data following the first RSEG directive is placed in a relocatable segment called table.

The code following the second RSEG directive is placed in a relocatable segment called code:

```

        EXTERN    subrtn, divrtn
        RSEG      table
functable:                ; place the external symbols in a table
        DC32      subrtn, divrtn

        RSEG      code
        PUBLIC    main
main:
    jsri.l  functable    ; call the two functions in functable
    jsri.l  functable+4
    rts

```

Beginning a common segment

The following example defines two common segments containing variables:

```

        NAME      common1
        COMMON    data
count    DC32      1
        ENDMOD

```

```

NAME      common2
COMMON    data
up        DS8      1
          DS8      2
down      DS8      1
          END
```

Because the common segments have the same name, `data`, the variables `up` and `down` refer to the same locations in memory as the first and last bytes of the 4-byte variable `count`.

Aligning a segment

This example starts a relocatable segment, moves to an even address, and adds some data. It then aligns to a 64-byte boundary, filling any intermediate space with `0xFF`, before creating a 64-byte table.

```

RSEG      data      ; start a relocatable segment
EVEN                      ; make sure the data is word-aligned
target    DC16 1
best      DC16 1
          ALIGN      6,0xFF ; Now align to a 64-byte boundary,
                           ; padding with 0xFF
results   DS8 64      ; and create a 64 byte table
```

Value assignment directives

These directives are used for assigning values to symbols.

Directive	Description
<code>=, EQU</code>	Assigns a permanent value local to a module.
<code>ASSIGN, SET, VAR</code>	Assigns a temporary value.
<code>DEFINE</code>	Defines a file-wide value.
<code>LIMIT</code>	Checks a value against limits.

Table 18: Value assignment directives

SYNTAX

```

label = expr
label ASSIGN expr
label DEFINE expr
label EQU expr
LIMIT expr, min, max, message
label SET expr
```

```
label VAR expr
```

PARAMETERS

<i>expr</i>	Value assigned to symbol or value to be tested.
<i>label</i>	Symbol to be defined.
<i>message</i>	A text message that will be printed when <i>expr</i> is out of range.
<i>min, max</i>	The minimum and maximum values allowed for <i>expr</i> .

DESCRIPTIONS

Defining a temporary value

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

Defining a permanent local value

Use `EQU` or `=` to assign a value to a symbol.

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 and all modules following that module in the same source file. If a `DEFINE` directive is placed outside of a module, the symbol will be known to all modules following the directive in the same source file.

A symbol which has been 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.

Checking symbol values

Use `LIMIT` to check that expressions lie within a specified range. If the expression is assigned a value outside the range, an error message will appear.

The check will occur as soon as the expression is resolved, which will be during linking if the expression contains external references. The *min* and *max* expressions cannot

involve references to forward or external labels, that is they must be resolved when encountered.

EXAMPLES

Redefining a symbol

The following example uses `SET` to redefine the symbol `cons` in a `REPT` loop to generate a table of the first 8 powers of 3:

```

NAME      table

; Generate table of powers of 3

cons      SET      1

cr_tabl   MACRO    times
          DC16     cons
cons      SET      cons*3
          IF       times>1
          cr_tabl  times-1
          ENENDIF
          ENDM
          RSEG     CODE

table:
          cr_tabl  4
          END      table

```

Using local and global symbols

In the following example the symbol `value` defined in module `add12` is local to that module; a distinct symbol of the same name is defined in module `add20`. The `DEFINE` directive is used for declaring `SR0` for use anywhere in the file:

```

NAME      add1
PUBLIC    add12
SR0       DEFINE   0x400
value     EQU      12
          RSEG     CODE

add12:
          ADD.W    #value,SR0
          RTS
          ENDMOD

NAME      add2
PUBLIC    add20
value     EQU      20

```

```

                                RSEG      CODE
add20:
                                ADD.W     #value,SR0
                                RTS
                                RETURN

                                END
```

The symbol SR0 defined in module add12 is also available to module add20.

Using the LIMIT directive

The following example sets the value of a variable called speed and then checks it, at assembly time, to see if it is in the range 10 to 30. This might be useful if speed is often changed at compile time, but values outside a defined range would cause undesirable behavior.

```

speed      SET      23
           LIMIT    speed,10,30, "Speed is out of range!"
```

Conditional assembly directives

These directives provide logical control over the selective assembly of source code. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

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

Table 19: Conditional assembly directives

SYNTAX

```

ELSE
ELSEIF condition
ENDIF
```

`IF condition`

PARAMETERS

condition One of the following:

An absolute expression	The expression must not contain forward or external references, and any non-zero value is considered as true.
<code>string1==string2</code>	The condition is true if <i>string1</i> and <i>string2</i> have the same length and contents.
<code>string1!=string2</code>	The condition is true if <i>string1</i> and <i>string2</i> have different length or contents.

DESCRIPTIONS

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

Use `ELSEIF` to introduce a new condition after an `IF` directive. Conditional assembly directives may 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 may 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 may be nested to any level.

EXAMPLES

The following macro adds a constant to a byte variable in memory:

```
addmac  MACRO    a,b          ; a should be a general operand,
                                ; b should be a literal
        IF      b=1
        INC.W   a
        ELSE
        ADD.W   #b,a
```

```
ENDIF
ENDM
```

If the argument to the macro is 1, it generates an `INC` instruction; otherwise it generates an `ADD` instruction.

It could be tested with the following program:

```
SR0      DEFINE    0x400
SR1      DEFINE    0x402
          RSEG      CODE

main:
          MOV.B     #0x0F, SR0
          addmac    SR0, 0x12
          addmac    SR1, 1
          RTS

          END
```

Macro processing directives

These directives allow user macros to be defined. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
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 20: Macro processing directives

SYNTAX

```
ENDM
ENDR
EXITM
```

```

LOCAL symbol [, symbol] ...
name MACRO [argument] [, argument] ...
REPT expr
REPTC formal, actual
REPTI formal, actual [, actual] ...

```

PARAMETERS

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

DESCRIPTIONS

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

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

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

Defining a macro

You define a macro with the statement:

```
name MACRO [argument] [, argument] ...
```

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

For example, you could define a macro `errmac` as follows:

```

errmac    EXTERN  abort
          MACRO   text
          MOVA    $+9, A0
          JMP     abort

```

```
DC8      text
ENDM
```

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

```
errmac   'Disk not ready'
```

The assembler will expand this to:

```
JSR      abort
DC8      'Disk not ready',0
```

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

The previous example could therefore be written as follows:

```
errmac   MACRO
          JSR      abort
          DC8      \1,0
          ENDM
```

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

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

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

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

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

Passing special characters

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

For example:

```
macld    MACRO   op
          MOV.L   op
          ENDM
```

The macro can be called using the macro quote characters:

```
macld    <0x800, 0x900>
END
```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 28.

Predefined macro symbols

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

```
FILL  MACRO
        IF _args == 2
            REPT  \1
            DC8  \2
        ENDR
        ELSE
            DC8  \1
        ENDIF
    ENDM

RSEG  CODE

        FILL  3, 4
        FILL  3

    END
```

How macros are processed

There are three distinct phases in the macro process:

- 1 The assembler performs scanning and saving of 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 will continue to be the output from the macro processor, until all lines of the current macro definition have been read.

Repeating statements

Use the `REPT . . . ENDR` structure to assemble the same block of instructions a number of times. If *expr* evaluates to 0 nothing will be generated.

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

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

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

EXAMPLES

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

Coding inline for efficiency

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

The following example outputs bytes from a buffer to a port:

```

NAME      play

portb     VAR      0x18
          RSEG     DATA
buffer    DS       256

          RSEG     CODE
play      MOVA     buffer,A0
          MOVA     portb,A1
          MOV.L    #255,R7R5
          SOUT.B
          RTS
          END
```

The main program calls this routine as follows:

```
doplay    JSR      play
```

For efficiency we can recode this using a macro:

```

NAME      play

portb     VAR      0x18
          RSEG     DATA
buffer    DS       256

play      MACRO
          MOVA     buffer,A0
```

```
        MOVA    portb,A1
        MOV.L   #255,R7R5
        SOUT.B
        ENDM

        RSEG    CODE
        play
        END
```

Using REPTC and REPTI

The following example assembles a series of calls to a subroutine `plotc` to plot each character in a string:

```
        NAME    reptc

        EXTERN  plotc
banner  REPTC   chr, "Welcome"

        MOV.B   #'chr',R0L      ; Pass char in R0L as a parameter
        JSR     plotc
        ENDR

        END
```

The following example uses `REPTI` to clear a number of memory locations:

```
        NAME    repti

        EXTERN  base, count, init
banner  REPTI   adds, base, count, init

        MOV.L   #0, adds
        ENDR

        END
```

Listing control directives

These directives provide control over the assembler list file.

Directive	Description
LSTCND	Controls conditional assembly listing.
LSTCOD	Controls multi-line code listing.

Table 21: Listing control directives

Directive	Description
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 21: Listing control directives (Continued)

SYNTAX

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

DESCRIPTIONS

Turning the listing on or off

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

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

Listing conditional code and strings

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

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

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

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

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

Controlling the listing of macros

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

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

Controlling the listing of generated lines

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

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

Generating a cross-reference table

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

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

EXAMPLES

Turning the listing on or off

To disable the listing of a debugged section of program:

```
LSTOUT-
; Debugged section
LSTOUT+
; Not yet debugged
```

Listing conditional code and strings

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

```
NAME      lstcndtst
EXTERN    print

RSEG      prom

debug     SET      0
begin     IF        debug
          JSR       print
        ENDIF

LSTCND+
```

```
begin2 IF      debug
      JSR      print
      ENDIF

      END
```

Controlling the listing of macros

The following example shows the effect of LSTMAC and LSTEXP:

```
dec2      MACRO  arg
          DEC.L  arg
          DEC.L  arg
          ENDM

          LSTMAC-
inc2      MACRO  arg
          INC.L  arg
          INC.L  arg
          ENDM

          EXTERN memlock
begin     dec2   memlock
          LSTEXP-
          inc2   memlock
          RTS

          END    begin
```

C-style preprocessor directives

The following 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.

Table 22: C-style preprocessor directives

Directive	Description
#include	Includes a file.
#line	Changes the line numbers of the source code lines immediately following the #line directive, or the filename of the file being compiled.
#pragma	Controls extension features. The supported #pragma directives are described in the chapter <i>Pragma directives</i> .
#undef	Undefines a preprocessor symbol.

Table 22: C-style preprocessor directives (Continued)

SYNTAX

```
#define symbol text
#elif condition
#else
#endif
#error "message"
#if condition
#ifdef symbol
#ifndef symbol
#include {"filename" | <filename>}
#undef symbol
```

PARAMETERS

<i>condition</i>	An absolute expression	The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true.
<i>filename</i>	Name of file to be included.	
<i>message</i>	Text to be displayed.	
<i>symbol</i>	Preprocessor symbol to be defined, undefined, or tested.	
<i>text</i>	Value to be assigned.	

DESCRIPTIONS

It is important to avoid mixing the assembler language with the C-style preprocessor directives. Conceptually, they are different languages and mixing them may 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 will not be available during the preprocessing phase.

Defining and undefining preprocessor symbols

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

```
#define symbol value
```

is similar to:

```
symbol ASSIGN 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 a `#endif` or `#else` directive is found.

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

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

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

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

Including source files

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

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

- If the name of the `#include` file is an absolute path, that file is opened.
- When the assembler encounters the name of an `#include` file in angle brackets such as:

```
#include <ior32c121.h>
```

it searches the following directories for the file to include:

- 1 The directories specified with the `-I` option, in the order that they were specified.
- 2 The directories specified using the `AR32C_INC` environment variable, if any.

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

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

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

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

Use angle brackets for header files provided with the R32C IAR Assembler, and double quotes for header files that are part of your application.

Displaying errors

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

Comments in C-style preprocessor directives

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

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

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

The following expression will evaluate to 3 because the comment character will be preserved by `#define`:

```
#define x 3      ; comment
exp      EQU x*8+5
```

The following 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 */

MOV.W    #five,R5          ; syntax error!
; Expands to "MOV #5 ; this comment is not OK,R5"

MOV.W    #six+seven,R6     ; OK
; Expanded to "MOV #6+7,R6"
```

EXAMPLES

Using conditional preprocessor directives

The following example defines the labels `tweak` and `adjust`. If `adjust` is defined, then register `A0` is decremented by an amount that depends on `adjust`, in this case 30.

```
#define tweak 1
#define adjust 3

#ifdef tweak
#if adjust==1
    ADD.L    #-4,A0
#elif adjust==2
    ADD.L    #-20,A0
#elif adjust==3
    ADD.L    #-30,A0
#endif
#endif                                     /* ifdef tweak */
```

Including a source file

The following example uses `#include` to include a file defining macros into the source file. For example, the following macros could be defined in `macros.s53`:

```
; exchange a and b using c as temporary
xch    MACRO    a,b,c
        MOV.W    a,c
        MOV.W    b,a
        MOV.W    c,b
    ENDM
```

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

```

NAME      include

SR0       DEFINE  0x400
SR1       DEFINE  0x402
SR2       DEFINE  0x404

; standard macro definitions

#include "macros.s53"

; program

main:
    xch     SR0, SR1, SR2
    RTS
    END     main

```

Data definition or allocation directives

These directives define values or reserve memory. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description
DC8	Generates 8-bit constants, including strings.
DC16	Generates 16-bit constants.
DC24	Generates 24-bit constants.
DC32	Generates 32-bit constants.
DC64	Generates 64-bit constants.
DF32	Generates 32-bit floating-point constants.
DF64	Generates 64-bit floating-point constants.
DQ15	Generates 16-bit fractional constants.
DQ31	Generates 32-bit fractional constants.
DS8	Allocates space for 8-bit integers.
DS16	Allocates space for 16-bit integers.
DS24	Allocates space for 24-bit integers.
DS32	Allocates space for 32-bit integers.

Table 23: Data definition or allocation directives

Directive	Description
DS64	Allocates space for 64-bit integers.

Table 23: Data definition or allocation directives (Continued)

SYNTAX

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

PARAMETERS

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

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

DESCRIPTIONS

Use the data definition and allocation directives according to the following table; it shows which directives reserve and initialize memory space or reserve uninitialized memory space, and their size.

Size	Reserve and initialize memory	Reserve uninitialized memory
8-bit integers	DC8	DS8

Table 24: Using data definition or allocation directives

Size	Reserve and initialize memory	Reserve uninitialized memory
16-bit integers	DC16	DS16
24-bit integers	DC24	DS24
32-bit integers	DC32	DS32
64-bit integers	DC64	DS64
32-bit floats	DF32	DS32
64-bit floats	DF64	DS64
16-bit fractionals	DQ15	DS16
32-bit fractionals	DQ31	DS32

Table 24: Using data definition or allocation directives (Continued)

EXAMPLES

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 0xA bytes:

```
table    DS8    0xA
```

Assembler control directives

These directives provide control over the operation of the assembler. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
<code>/*comment*/</code>	C-style comment delimiter.	
<code>//</code>	C++style comment delimiter.	
<code>CASEOFF</code>	Disables case sensitivity.	

Table 25: Assembler control directives

Directive	Description	Expression restrictions
CASEON	Enables case sensitivity.	
RADIX	Sets the default base on all numeric values.	No forward references No external references Absolute Fixed

Table 25: Assembler control directives (Continued)

SYNTAX

*/*comment*/*
//comment
CASEOFF
CASEON
RADIX *expr*

PARAMETERS

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

DESCRIPTIONS

Use */* . . . */* to comment sections of the assembler listing.
Use *//* to mark the rest of the line as comment.
Use 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 XLINK should be written in upper case in the XLINK definition file.

EXAMPLES

Defining comments

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

```
/*
Program to read serial input.
Version 1: 19.2.07
Author: mjp
*/
```

See also, *Comments in C-style preprocessor directives*, page 79.

Changing the base

To set the default base to 16:

```
RADIX 16D
MOV.L #12,R2R0
```

The immediate argument will then be interpreted as the hexadecimal constant 12, that is decimal 18.

To reset the base from 16 to 10 again, the argument must be written in hexadecimal format, for example:

```
RADIX 0x0A
```

Controlling case sensitivity

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

```
label NOP                                ; Stored as "LABEL"
JMP LABEL

; The following will generate a
; duplicate label error:

CASEOFF
label NOP
LABEL NOP                                ; Error, "LABEL" already defined

END
```

Call frame information directives

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

Directive	Description
CFI BASEADDRESS	Declares a base address CFA (Canonical Frame Address).
CFI BLOCK	Starts a data block.
CFI CODEALIGN	Declares code alignment.
CFI COMMON	Starts or extends a common block.
CFI CONDITIONAL	Declares data block to be a conditional thread.
CFI DATAALIGN	Declares data alignment.
CFI ENDBLOCK	Ends a data block.
CFI ENDCOMMON	Ends a common block.
CFI ENDNAMES	Ends a names block.
CFI FRAMECELL	Creates a reference into the caller's frame.
CFI FUNCTION	Declares a function associated with data block.
CFI INVALID	Starts range of invalid backtrace information.
CFI NAMES	Starts a names block.
CFI NOFUNCTION	Declares data block to not be associated with a function.
CFI PICKER	Declares data block to be a picker thread.
CFI REMEMBERSTATE	Remembers the backtrace information state.
CFI RESOURCE	Declares a resource.
CFI RESOURCEPARTS	Declares a composite resource.
CFI RESTORESTATE	Restores the saved backtrace information state.
CFI RETURNADDRESS	Declares a return address column.
CFI STACKFRAME	Declares a stack frame CFA.
CFI STATICOVERLAYFRAME	Declares a static overlay frame CFA.
CFI VALID	Ends range of invalid backtrace information.
CFI VIRTUALRESOURCE	Declares a virtual resource.
CFI <i>cfa</i>	Declares the value of a CFA.
CFI <i>resource</i>	Declares the value of a resource.

Table 26: Call frame information directives

SYNTAX

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

Names block directives

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

Extended names block directives

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

Common block directives

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

Extended common block directives

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

Data block directives

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

PARAMETERS

<i>bits</i>	The size of the resource in bits.
<i>cell</i>	The name of a frame cell.
<i>cfa</i>	The name of a CFA (canonical frame address).
<i>cfiexpr</i>	A CFI expression (see <i>CFI expressions</i> , page 95).
<i>codealignfactor</i>	The smallest factor of all instruction sizes. Each CFI directive for a data block must be placed according to this alignment. 1 is the default and can always be used, but a larger value will shrink the produced backtrace information in size. The possible range is 1–256.
<i>commonblock</i>	The name of a previously defined common block.
<i>constant</i>	A constant value or an assembler expression that can be evaluated to a constant value.
<i>dataalignfactor</i>	The smallest factor of all frame sizes. If the stack grows towards higher addresses, the factor is negative; if it grows towards lower addresses, the factor is positive. 1 is the default, but a larger value will shrink the produced backtrace information in size. The possible ranges are -256 to -1 and 1 to 256.
<i>label</i>	A function label.
<i>name</i>	The name of the block.

<i>namesblock</i>	The name of a previously defined names block.
<i>offset</i>	The offset relative the CFA. An integer with an optional sign.
<i>part</i>	A part of a composite resource. The name of a previously declared resource.
<i>resource</i>	The name of a resource.
<i>segment</i>	The name of a segment.
<i>size</i>	The size of the frame cell in bytes.
<i>type</i>	The memory type, such as <code>CODE</code> , <code>CONST</code> or <code>DATA</code> . In addition, any of the memory types supported by the IAR XLINK Linker. It is used solely for the purpose of denoting an address space.

DESCRIPTIONS

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

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

Backtrace rows and columns

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

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

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

Defining a names block

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

Start and end a names block with the directives:

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

where *name* is the name of the block.

Only one names block can be open at a time.

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

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

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

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

More than one resource can be declared by separating them with commas.

A resource may 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 segment type (to get the address space). More than one stack frame CFA can be declared by separating them with commas.

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

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

```
CFI STATICOVERLAYFRAME cfa segment
```

The parameters are the name of the CFA and the name of the segment where the static overlay for the function is located. More than one static overlay frame CFA can be declared by separating them with commas.

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

```
CFI BASEADDRESS cfa type
```

The parameters are the name of the CFA and the segment type. More than one base address CFA can be declared by separating them with commas.

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

Extending a names block

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

Extend an existing names block with the directive:

```
CFI NAMES name EXTENDS namesblock
```

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

```
CFI ENDNAMES name
```

Defining a common block

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

Start a common block with the directive:

```
CFI COMMON name USING namesblock
```

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

Declare the return address column with the directive:

```
CFI RETURNADDRESS resource type
```

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

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

Inside a common block you can declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 87. For more information on these directives, see *Simple rules*, page 93, and *CFI expressions*, page 95.

Extending a common block

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

Extend an existing common block with the directive:

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

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

```
CFI ENDCOMMON name
```

Defining a data block

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

Start a data block with the directive:

```
CFI BLOCK name USING commonblock
```

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

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

```
CFI FUNCTION label
```

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

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

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

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

Inside a data block you may manipulate the values of the columns by using the directives listed last in *Data block directives*, page 88. For more information on these directives, see *Simple rules*, page 93, and *CFI expressions*, page 95.

SIMPLE RULES

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

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

These simple rules can be used 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, a full CFI expression can be used to describe the information (see *CFI expressions*, page 95). However, whenever possible, you should always use a simple rule instead of a CFI expression.

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

Simple rules for resources

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

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

```
CFI REG SAMEVALUE
```

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

```
CFI REG UNDEFINED
```

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

```
CFI REG1 REG2
```

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

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

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

```
CFI RET CONCAT
```

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

Simple rules for CFAs

In contrast with the rules for resources, the rules for CFAs describe the address of the beginning of the call frame. The call frame often includes the return address pushed by the subroutine calling instruction. The CFA rules describe how to compute the address to the beginning of the current call frame. There are two different forms of CFAs, stack frames and static overlay frames, each declared in the associated names block. See *Names block directives*, page 87.

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

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

```
CFI CFA_SP NOTUSED
```

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

```
CFI CFA_SP SP + 4
```

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

CFI EXPRESSIONS

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

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

In the operand descriptions, *cfiexpr* denotes one of the following:

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

Unary operators

Overall syntax: *OPERATOR*(*operand*)

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

Table 27: Unary operators in CFI expressions

Binary operators

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

Operator	Operands	Description
ADD	<i>cfiexpr</i> , <i>cfiexpr</i>	Addition
AND	<i>cfiexpr</i> , <i>cfiexpr</i>	Bitwise AND
DIV	<i>cfiexpr</i> , <i>cfiexpr</i>	Division
EQ	<i>cfiexpr</i> , <i>cfiexpr</i>	Equal

Table 28: Binary operators in CFI expressions

Operator	Operands	Description
GE	<i>cfiexpr,cfiexpr</i>	Greater than or equal
GT	<i>cfiexpr,cfiexpr</i>	Greater than
LE	<i>cfiexpr,cfiexpr</i>	Less than or equal
LSHIFT	<i>cfiexpr,cfiexpr</i>	Logical shift left of the left operand. The number of bits to shift is specified by the right operand. The sign bit will not be preserved when shifting.
LT	<i>cfiexpr,cfiexpr</i>	Less than
MOD	<i>cfiexpr,cfiexpr</i>	Modulo
MUL	<i>cfiexpr,cfiexpr</i>	Multiplication
NE	<i>cfiexpr,cfiexpr</i>	Not equal
OR	<i>cfiexpr,cfiexpr</i>	Bitwise OR
RSHIFTA	<i>cfiexpr,cfiexpr</i>	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 will be preserved when shifting.
RSHIFTL	<i>cfiexpr,cfiexpr</i>	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	<i>cfiexpr,cfiexpr</i>	Subtraction
XOR	<i>cfiexpr,cfiexpr</i>	Bitwise XOR

Table 28: Binary operators in CFI expressions (Continued)

Ternary operators

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

Operator	Operands	Description
FRAME	<i>cfa, size, offset</i>	Gets the value from a stack frame. The operands are: <i>cfa</i> An identifier denoting a previously declared CFA. <i>size</i> A constant expression denoting a size in bytes. <i>offset</i> A constant expression denoting an offset in bytes. Gets the value at address <i>cfa+offset</i> of size <i>size</i> .

Table 29: Ternary operators in CFI expressions

Operator	Operands	Description
IF	<i>cond, true, false</i>	Conditional operator. The operands are: <i>cond</i> A CFA expression denoting a condition. <i>true</i> Any CFA expression. <i>false</i> Any CFA expression. If the conditional expression is non-zero, the result is the value of the <i>true</i> expression; otherwise the result is the value of the <i>false</i> expression.
LOAD	<i>size, type, addr</i>	Gets the value from memory. The operands are: <i>size</i> A constant expression denoting a size in bytes. <i>type</i> A memory type. <i>addr</i> A CFA expression denoting a memory address. Gets the value at address <i>addr</i> in segment type <i>type</i> of size <i>size</i> .

Table 29: Ternary operators in CFI expressions (Continued)

EXAMPLE

The following is a generic example and not an example specific to the R32C/100 microcomputer. This will simplify the example and clarify the usage of the CFI directives. A target-specific example can be obtained by generating assembler output when compiling a C source file.

Consider a generic processor with a stack pointer *SP*, and two registers *R0* and *R1*. Register *R0* will be used as a scratch register (the register is destroyed by the function call), whereas register *R1* has to be restored after the function call. For reasons of simplicity, all instructions, registers, and addresses will have a width of 16 bits.

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

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

Table 30: Code sample with backtrace rows and columns

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

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

Defining the names block

The names block for the small example above would be:

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

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

Defining the common block

The common block for the simple example above would be:

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

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

Defining the data block

Continuing the simple example, the data block would be:

```
RSEG CODE:CODE
CFI BLOCK func1block USING trivialCommon
CFI FUNCTION func1
func1:
```

```
PUSH    R1
CFI      CFA SP + 4
CFI      R1 FRAME(CFA, -4)
MOV      R1, #4
CALL     func2
POP      R0
CFI      R1 R0
CFI      CFA SP + 2
MOV      R1, R0
CFI      R1 SAMEVALUE
RET
CFI ENDBLOCK func1block
```

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

Pragma directives

This chapter describes the pragma directives of the R32C IAR Assembler.

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

Summary of pragma directives

The following table shows the pragma directives of the assembler:

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

Table 31: Pragma directives summary

Descriptions of pragma directives

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

```
#pragma pragmaname=pragmavalue
```

or

```
#pragma pragmaname = pragmavalue
```

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

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

```
#pragma diag_default=Pe117
```

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

<code>#pragma diag_error</code>	<hr/> <code>#pragma diag_error=tag, tag, ...</code> <p>Changes the severity level to <code>error</code> for the specified diagnostics. For example:</p> <pre>#pragma diag_error=Pe117</pre> <p>See the chapter <i>Diagnostics</i> for more information about diagnostic messages.</p>
<code>#pragma diag_remark</code>	<hr/> <code>#pragma diag_remark=tag, tag, ...</code> <p>Changes the severity level to <code>remark</code> for the specified diagnostics. For example:</p> <pre>#pragma diag_remark=Pe177</pre> <p>See the chapter <i>Diagnostics</i> for more information about diagnostic messages.</p>
<code>#pragma diag_suppress</code>	<hr/> <code>#pragma diag_suppress=tag, tag, ...</code> <p>Suppresses the diagnostic messages with the specified tags. For example:</p> <pre>#pragma diag_suppress=Pe117, Pe177</pre> <p>See the chapter <i>Diagnostics</i> for more information about diagnostic messages.</p>
<code>#pragma diag_warning</code>	<hr/> <code>#pragma diag_warning=tag, tag, ...</code> <p>Changes the severity level to <code>warning</code> for the specified diagnostics. For example:</p> <pre>#pragma diag_warning=Pe826</pre> <p>See the chapter <i>Diagnostics</i> for more information about diagnostic messages.</p>
<code>#pragma message</code>	<hr/> <code>#pragma message(string)</code> <p>Makes the assembler print a message on <code>stdout</code> when the file is assembled. For example:</p> <pre>#ifdef TESTING #pragma message("Testing") #endif</pre>

Diagnostics

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

Message format

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

```
filename,linenumber level[tag]: message
```

where *filename* is the name of the source file in which the error was encountered; *linenumber* is the line number at which the assembler detected the error; *level* is the level of seriousness of the diagnostic; *tag* is a unique tag that identifies the diagnostic message; *message* is a self-explanatory message, possibly several lines long.



Diagnostic messages are displayed on the screen, as well as 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 32.

Warning

A diagnostic message that is produced when the assembler finds a programming error or omission which is of concern but not so severe as to prevent the completion of compilation. Warnings can be disabled by use of the command-line option `--no_warnings`, see `--no_warnings`, page 29.

Error

A diagnostic message that is produced when the assembler has found a construct which clearly violates the language rules, such that code cannot be produced. An error will produce a non-zero exit code.

Fatal error

A diagnostic message that is produced when the assembler has found a condition that not only prevents code generation, but which makes further processing of the source code pointless. After the diagnostic has been issued, compilation terminates. A fatal error will produce a non-zero exit code.

SETTING THE SEVERITY LEVEL

The diagnostic messages can be suppressed or the severity level can be changed for all types of diagnostics except for fatal errors and some of the regular errors.

See *Summary of assembler options*, page 19, for a description of the assembler options that are available for setting severity levels.

See the chapter *Pragma directives*, for a description of the pragma directives that are available for setting severity levels.

INTERNAL ERROR

An internal error is a diagnostic message that signals that there has been a serious and unexpected failure due to a fault in the assembler. It is produced using the following form:

`Internal error: message`

where *message* is an explanatory message. If internal errors occur, they should be reported to your software distributor or IAR 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

absolute expressions	12
absolute segments	60
ADD (CFI operator)	95
address field, in assembler list file	13
ALIGN (assembler directive)	58
alignment, of segments	61
ALIGNRAM (assembler directive)	58
AND (assembler operator)	38
AND (CFI operator)	95
architecture, R32C/100	ix
_args (predefined macro symbol)	71
__AR32C__ (predefined symbol)	10
AR32C_INC (environment variable)	4
ASCII character constants	7
ASEG (assembler directive)	58
ASEGN (assembler directive)	58
asm (filename extension)	3
ASMR32C (environment variable)	4
assembler BLOCK (assembler directive)	56
assembler control directives	83
assembler diagnostics	103
assembler directives	
ALIGN	58
ALIGNRAM	58
ASEG	58
ASEGN	58
assembler control	83
ASSIGN	63
call frame information	86
CASEOFF	83
CASEON	84
CFI directives	86
COMMON	58
conditional assembly	66
<i>See also</i> C-style preprocessor directives	
C-style preprocessor	76
data definition or allocation	81
DC8	81
DC16	81
DC24	81
DC32	81
DC64	81
DEFINE	63
DF32	81
DF64	81
DQ15	81
DQ31	81
DS8	81
DS16	81
DS24	81
DS32	81
DS64	82
ELSE	66
ELSEIF	66
END	53
ENDIF	66
ENDM	68
ENDMOD	53
ENDR	68
EQU	63
EVEN	59
EXITM	68
IF	66
LIBRARY	53
LIMIT	63
list file control	73
LOCAL	68
LSTCND	73
LSTCOD	73
LSTEXP	74
LSTMAC	74
LSTOUT	74
LSTREP	74
LSTXRF	74
MACRO	68
macro processing	68

MODULE	53	assembler IMPORT (assembler directive)	56
module control	52	assembler instructions	5
NAME	53	assembler labels	8
ODD	59	format of	5
ORG	59	assembler list files	
PROGRAM	53	address field	13
PUBLIC	56	comments	84
PUBWEAK	56	conditional code and strings	74
RADIX	84	cross-references, generating	27, 75
REPT	68	data field	13
REPTC	68	disabling	74
REPTI	68	enabling	74
REQUIRE	56	filename, specifying	27
RSEG	59	generated lines, controlling	75
RTMODEL	53	macro-generated lines, controlling	75
segment control	58	symbol and cross-reference table	14
SET	63	assembler macros	
summary	49	arguments, passing to	71
SYMBOL	56	defining	69
symbol control	56	generated lines, controlling in list file	75
value assignment	63	in-line routines	72
VAR	63	predefined symbol	71
#define	76	processing	71
#elif	76	quote characters, specifying	28
#else	76	special characters, using	70
#endif	76	assembler object file	
#error	76	specifying filename	29
#if	76	assembler operators	33
#ifdef	76	AND	38
#ifndef	76	BINAND	38
#include	77	BINNOT	38
#line	77	BINOR	39
#pragma	77, 101	BINXOR	39
#undef	77	BYTE1	39
/*...*/	83	BYTE2	39
//	83	BYTE3	40
=	63	BYTE4	40
assembler environment variables	4	DATE	40
assembler EXTERN (assembler directive)	56	EQ	40

GE	41	>	41
GT	41	>=	41
HIGH	41	>>	45
HWRD	41	?:	38
in expressions	6	^	39
LE	42		39
LOW	42		43
LT	42	~	38
LWRD	42	assembler options	
MOD	43	passing to assembler	3
NE	43	specifying parameters	18
NOT	43	summary	19
OR	43	typographic convention	xi
precedence	33	-D	21
SFB	44	-f	26
SFE	44	-I	27
SHL	45	-l	27
SHR	45	-M	28
SIZEOF	45	-o	29
UGT	46	-r	31
ULT	46	--case_insensitive	20
UPPER	46	--debug	22
XOR	47	--dependencies	22
!	43	--diagnostics_tables	25
!=	43	--diag_error	24
%	43	--diag_remark	24
&	38	--diag_suppress	24
&&	38	--diag_warning	24
()	36	--dir_first	25
*	36	--double	25
+	36–37	--enable_multibytes	26
-	37	--error_limit	26
/	37	--header_context	26
<	42	--mnem_first	29
<<	45	--no_path_in_file_macros	29
<=	42	--no_warnings	29
<>	43	--no_wrap_diagnostics	29
=	40	--only_stdout	30
==	40	--preinclude	30

--preprocess	30
--remarks	32
--silent	32
--warnings_affect_exit_code	4, 32
--warnings_are_errors	32
assembler output, including debug information	22, 31
assembler OVERLAY (assembler directive)	56
assembler source files, including	78
assembler source format	5
assembler subversion number	11
assembler symbols	8
exporting	57
importing	57
in relocatable expressions	12
local	65
predefined	10
redefining	65
ASSIGN (assembler directive)	63
assumptions (programming experience)	ix

B

backtrace information, defining	86
BINAND (assembler operator)	38
BINNOT (assembler operator)	38
BINOR (assembler operator)	39
BINXOR (assembler operator)	39
BLOCK (assembler directive)	56
__BUILD_NUMBER__ (predefined symbol)	10
BYTE1 (assembler operator)	39
BYTE2 (assembler operator)	39
BYTE3 (assembler operator)	40
BYTE4 (assembler operator)	40

C

call frame information directives	86
case sensitivity, controlling	20, 84
CASEOFF (assembler directive)	83

CASEON (assembler directive)	84
--case_insensitive (assembler option)	20
CFI directives	86
CFI expressions	95
CFI operators	95
character constants, ASCII	7
code models	
identifying (__CODE_MODEL__)	10
specifying on command line (--code_model)	21
__CODE_MODEL__ (predefined symbol)	10
--code_model (assembler option)	21
command line options	
part of invocation syntax	2
passing	3
command line, extending	26
comments	
in assembler list file	84
in assembler source code	5
multi-line, using with assembler directives	85
comments, in C-style preprocessor directives	79
common segments	61
COMMON (assembler directive)	58
compiler, invocation syntax	2
COMPLEMENT (CFI operator)	95
computer style, typographic convention	xi
conditional assembly directives	66
<i>See also</i> C-style preprocessor directives	
conditional code and strings, listing	74
constants, default base of	84
constants, integer	6
conventions, typographic	xi
CRC, in assembler list file	14
cross-references, in assembler list file	27, 75
C-style preprocessor directives	76

D

-D (assembler option)	21
data allocation directives	81

data definition directives 81
 data field, in assembler list file 13
 --data_model (assembler option). 22
 __DATA_MODEL__ (predefined symbol) 11
 __DATE__ (predefined symbol) 11
 DATE (assembler operator). 40
 DC8 (assembler directive). 81
 DC16 (assembler directive). 81
 DC24 (assembler directive). 81
 DC32 (assembler directive). 81
 DC64 (assembler directive). 81
 --debug (assembler option) 22
 debug information, including in assembler output . . . 22, 31
 default base, for constants 84
 #define (assembler directive) 76
 DEFINE (assembler directive) 63
 --dependencies (assembler option) 22
 DF32 (assembler directive). 81
 DF64 (assembler directive). 81
 diagnostic messages 103
 classifying as errors 24
 classifying as remarks 24
 classifying as warnings 24
 disabling warnings 29
 disabling wrapping of 29
 enabling remarks 32
 listing all 25
 suppressing 24
 --diagnostics_tables (assembler option) 25
 diag_default (#pragma directive) 101
 --diag_error (assembler option). 24
 diag_error (#pragma directive) 102
 --diag_remark (assembler option) 24
 diag_remark (#pragma directive) 102
 --diag_suppress (assembler option). 24
 diag_suppress (#pragma directive) 102
 --diag_warning (assembler option) 24
 diag_warning (#pragma directive) 102
 directives. *See* assembler directives

--dir_first (assembler option) 25
 DIV (CFI operator). 95
 document conventions. xi
 __DOUBLE__ (predefined symbol) 11
 --double (assembler option) 25
 DQ15 (assembler directive) 81
 DQ31 (assembler directive) 81
 DS8 (assembler directive). 81
 DS16 (assembler directive). 81
 DS24 (assembler directive). 81
 DS32 (assembler directive). 81
 DS64 (assembler directive). 82

E

efficient coding techniques 14
 #elif (assembler directive). 76
 #else (assembler directive) 76
 ELSE (assembler directive). 66
 ELSEIF (assembler directive). 66
 --enable_multibytes (assembler option) 26
 END (assembler directive) 53
 #endif (assembler directive) 76
 ENDIF (assembler directive) 66
 ENDM (assembler directive) 68
 ENDMOD (assembler directive). 53
 ENDR (assembler directive). 68
 environment variables
 AR32C_INC 4
 ASMR32C 4
 assembler 4
 EQ (assembler operator). 40
 EQ (CFI operator). 95
 EQU (assembler directive) 63
 #error (assembler directive) 76
 error messages 103
 classifying 24
 #error, using to display. 79
 --error_limit (assembler option) 26

EVEN (assembler directive)	59
EXITM (assembler directive)	68
experience, programming	ix
expressions	6
expressions. <i>See</i> assembler expressions	
extended command line file (extend.xcl).	26
EXTERN (assembler directive).	56

F

-f (assembler option).	26
false value, in assembler expressions	8
fatal error messages	104
__FILE__ (predefined symbol).	11
file dependencies, tracking	22
file extensions. <i>See</i> filename extensions	
file types	
assembler source	3
extended command line	26
#include, specifying path	27
filename extensions	
asm	3
msa	3
s53	3
xcl	26
filenames, specifying for assembler output	29
filename, of object file	29
floating-point constants.	7
formats, assembler source code.	5
fractions	8
FRAME (CFI operator).	96

G

GE (assembler operator).	41
GE (CFI operator).	96
global value, defining	64
glossary.	ix
GT (assembler operator).	41

GT (CFI operator).	96
----------------------------	----

H

header files, SFR.	14
--header_context (assembler option).	26
HIGH (assembler operator).	41
HWRD (assembler operator)	41

I

-I (assembler option).	27
IAR Systems Technical Support	104
__IAR_SYSTEMS_ASM__ (predefined symbol)	11
#if (assembler directive)	76
IF (assembler directive)	66
IF (CFI operator).	97
#ifdef (assembler directive).	76
#ifndef (assembler directive).	76
immediate operands, specifying	15
#immex (operand encoding format)	15
IMPORT (assembler directive).	56
#include files, specifying	27
#include (assembler directive)	77
include paths, specifying.	27
instruction set, R32C/100	ix
integer constants	6
internal error.	104
invocation syntax	2
in-line coding, using macros	72

L

-l (assembler option).	27
labels. <i>See</i> assembler labels	
LE (assembler operator)	42
LE (CFI operator)	96
library modules.	54
LIBRARY (assembler directive).	53

LIMIT (assembler directive)	63
__LINE__ (predefined symbol)	11
#line (assembler directive)	77
list file format	13
body	13
CRC	14
header	13
symbol and cross reference	
listing control directives	73
LITERAL (CFI operator)	95
LOAD (CFI operator)	97
local value, defining	64
LOCAL (assembler directive)	68
location counter. <i>See</i> program location counter	
LOW (assembler operator)	42
LSHIFT (CFI operator)	96
LSTCND (assembler directive)	73
LSTCOD (assembler directive)	73
LSTEXP (assembler directives)	74
LSTMAC (assembler directive)	74
LSTOUT (assembler directive)	74
LSTREP (assembler directive)	74
LSTXRF (assembler directive)	74
LT (assembler operator)	42
LT (CFI operator)	96
LWRD (assembler operator)	42

M

-M (assembler option)	28
macro processing directives	68
macro quote characters	70
specifying	28
MACRO (assembler directive)	68
macros. <i>See</i> assembler macros	
memory space, reserving and initializing	82
memory, reserving space in	81
message (#pragma directive)	102
messages, excluding from standard output stream	32

--mnem_first (assembler option)	29
MOD (assembler operator)	43
MOD (CFI operator)	96
module consistency	55
module control directives	52
MODULE (assembler directive)	53
modules	
assembling multi-modules files	54
terminating	54
msa (filename extension)	3
MUL (CFI operator)	96
multibyte character support	26

N

NAME (assembler directive)	53
NE (assembler operator)	43
NE (CFI operator)	96
NOT (assembler operator)	43
NOT (CFI operator)	95
--no_path_in_file_macros (assembler option)	29
--no_warnings (assembler option)	29
--no_wrap_diagnostics (assembler option)	29

O

-o (assembler option)	29
ODD (assembler directive)	59
--only_stdout (assembler option)	30
operands	
format of	5
immediate	15
in assembler expressions	6
operations, format of	5
operation, silent	32
operators. <i>See</i> assembler operators	
option summary	19
OR (assembler operator)	43
OR (CFI operator)	96

ORG (assembler directive)	59
OVERLAY (assembler directive)	56

P

parameters	
specifying	18
typographic convention	xi
PLC. <i>See</i> program location counter	
#pragma (assembler directive)	77, 101
precedence, of assembler operators.	33
predefined register symbols	9
predefined symbols	10
in assembler macros.	71
__AR32C__	10
__BUILD_NUMBER__	10
__CODE_MODEL__	10
__DATA_MODEL__	11
__DATE__	11
__DOUBLE__	11
__FILE__	11
__IAR_SYSTEMS_ASM__	11
__LINE__	11
__SUBVERSION__	11
__TIME__	11
__VER__	11
--preinclude (assembler option)	30
--preprocess (assembler option)	30
preprocessor symbols	
defining and undefining	78
defining on command line	21
prerequisites (programming experience).	ix
program counter. <i>See</i> program location counter	
program location counter (PLC)	8
setting	61
program modules, beginning.	54
PROGRAM (assembler directive).	53
programming experience, required	ix
programming hints	14

PUBLIC (assembler directive)	56
PUBWEAK (assembler directive).	56

R

-r (assembler option).	31
RADIX (assembler directive)	84
reference information, typographic convention.	xi
registered trademarks	ii
registers	9
relocatable expressions	12
relocatable segments, beginning	60
remark (diagnostic message).	103
classifying	24
enabling	32
--remarks (assembler option)	32
repeating statements	71
REPT (assembler directive)	68
REPTC (assembler directive)	68
REPTI (assembler directive).	68
REQUIRE (assembler directive).	56
RSEG (assembler directive)	59
RSHIFTA (CFI operator)	96
RSHIFTL (CFI operator)	96
RTMODEL (assembler directive).	53
rules, in CFI directives	93
runtime model attributes, declaring.	55
R32C/100 architecture and instruction set	ix

S

segment control directives.	58
segments	
absolute	60
aligning	61
common, beginning	61
relocatable	60
SET (assembler directive).	63

severity level, of diagnostic messages	103
specifying	104
SFB (assembler operator)	44
SFE (assembler operator)	44
SFR. <i>See</i> special function registers	
SHL (assembler operator)	45
SHR (assembler operator)	45
--silent (assembler option)	32
silent operation, specifying	32
simple rules, in CFI directives	93
SIZEOF (assembler operator)	45
source files	
including	78
list all referred	26
source format, assembler	5
special function registers	14
standard error	30
standard output stream, disabling messages to	32
standard output, specifying	30
statements, repeating	71
stderr	30
stdout	30
SUB (CFI operator)	96
__SUBVERSION__ (predefined symbol)	11
Support, Technical	104
symbol and cross-reference table, in assembler list file.	14
<i>See also</i> Include cross-reference	
symbol control directives	56
symbol values, checking	64
SYMBOL (assembler directive)	56
symbols	
<i>See also</i> assembler symbols	
exporting to other modules	57
predefined, in assembler	10
predefined, in assembler macro	71
user-defined, case sensitive	20
syntax conventions	xi
s53 (filename extension)	3

T

Technical Support, IAR Systems	104
temporary values, defining	64
terminology	ix
__TIME__ (predefined symbol)	11
time-critical code	72
trademarks	ii
true value, in assembler expressions	8
typographic conventions	xi

U

UGT (assembler operator)	46
ULT (assembler operator)	46
UMINUS (CFI operator)	95
#undef (assembler directive)	77
UPPER (assembler operator)	46
user symbols, case sensitive	20

V

value assignment directives	63
values, defining	81
VAR (assembler directive)	63
__VER__ (predefined symbol)	11
version, of assembler	11

W

warnings	103
classifying	24
disabling	29
exit code	32
treating as errors	32
--warnings_affect_exit_code (assembler option)	4, 32
--warnings_are_errors (assembler option)	32

X

xcl (filename extension)	26
XOR (assembler operator)	47
XOR (CFI operator)	96

Symbols

! (assembler operator)	43
!= (assembler operator)	43
#define (assembler directive)	76
#elif (assembler directive)	76
#else (assembler directive)	76
#endif (assembler directive)	76
#error (assembler directive)	76
#if (assembler directive)	76
#ifdef (assembler directive)	76
#ifndef (assembler directive)	76
#immex (operand encoding format)	15
#include files, specifying	27
#include (assembler directive)	77
#line (assembler directive)	77
#pragma (assembler directive)	77, 101
#undef (assembler directive)	77
\$ (program location counter)	8
% (assembler operator)	43
& (assembler operator)	38
&& (assembler operator)	38
() (assembler operator)	36
* (assembler operator)	36
+ (assembler operator)	36–37
- (assembler operator)	37
-D (assembler option)	21
-f (assembler option)	26
-I (assembler option)	27
-l (assembler option)	27
-M (assembler option)	28
-o (assembler option)	29
-r (assembler option)	31

--case_insensitive (assembler option)	20
--code_model (assembler option)	21
--data_model (assembler option)	22
--debug (assembler option)	22
--dependencies (assembler option)	22
--diagnostics_tables (assembler option)	25
--diag_error (assembler option)	24
--diag_remark (assembler option)	24
--diag_suppress (assembler option)	24
--diag_warning (assembler option)	24
--dir_first (assembler option)	25
--double (assembler option)	25
--enable_multibytes (assembler option)	26
--error_limit (assembler option)	26
--header_context (assembler option)	26
--mnem_first (assembler option)	29
--no_path_in_file_macros (assembler option)	29
--no_warnings (assembler option)	29
--no_wrap_diagnostics (assembler option)	29
--only_stdout (assembler option)	30
--preinclude (assembler option)	30
--preprocess (assembler option)	30
--remarks (assembler option)	32
--silent (assembler option)	32
--warnings_affect_exit_code (assembler option)	4, 32
--warnings_are_errors (assembler option)	32
/ (assembler operator)	37
/*...*/ (assembler directive)	83
// (assembler directive)	83
< (assembler operator)	42
<< (assembler operator)	45
<= (assembler operator)	42
<> (assembler operator)	43
= (assembler directive)	63
= (assembler operator)	40
== (assembler operator)	40
> (assembler operator)	41
>= (assembler operator)	41
>> (assembler operator)	45

?: (assembler operator)	38
^ (assembler operator).	39
_args (predefined macro symbol)	71
__AR32C__ (predefined symbol)	10
__BUILD_NUMBER__ (predefined symbol)	10
__CODE_MODEL__ (predefined symbol).	10
__DATA_MODEL__ (predefined symbol)	11
__DATE__ (predefined symbol)	11
__DOUBLE__ (predefined symbol)	11
__FILE__ (predefined symbol).	11
__IAR_SYSTEMS_ASM__ (predefined symbol)	11
__LINE__ (predefined symbol)	11
__SUBVERSION__ (predefined symbol).	11
__TIME__ (predefined symbol)	11
__VER__ (predefined symbol)	11
! (assembler operator)	39
!! (assembler operator).	43
~ (assembler operator)	38