

# **ARM® IAR Assembler**

## **Reference Guide**

for Advanced RISC Machines Ltd's  
**ARM Cores**

## **COPYRIGHT NOTICE**

© Copyright 1999–2007 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.

ARM and Thumb are registered trademarks of Advanced RISC Machines Ltd.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

## **EDITION NOTICE**

Eighth edition: June 2007

Part number: AARM-8

This guide applies to version 5.x of the ARM® IAR Embedded Workbench® IDE.

Internal reference: ISUD.

# Contents

<b>Tables</b> .....	ix
<b>Preface</b> .....	xi
<b>Who should read this guide</b> .....	xi
<b>How to use this guide</b> .....	xi
<b>What this guide contains</b> .....	xii
<b>Other documentation</b> .....	xii
<b>Document conventions</b> .....	xiii
<b>Introduction to the ARM IAR Assembler</b> .....	1
<b>Introduction to assembler programming</b> .....	1
Getting started .....	2
<b>Modular programming</b> .....	2
<b>External interface details</b> .....	3
Assembler invocation syntax .....	3
Passing options .....	4
Environment variables .....	4
Error return codes .....	5
<b>Source format</b> .....	5
<b>Assembler instructions</b> .....	6
<b>Expressions, operands, and operators</b> .....	6
Integer constants .....	6
ASCII character constants .....	7
Floating-point constants .....	7
TRUE and FALSE .....	8
Symbols .....	8
Labels .....	9
Register symbols .....	9
Predefined symbols .....	10
Absolute and relocatable expressions .....	11
Expression restrictions .....	12

<b>List file format</b>	13
Header	13
Body	13
Summary	13
Symbol and cross-reference table	13
<b>Programming hints</b>	14
Accessing special function registers	14
Using C-style preprocessor directives	14
<b>Assembler options</b>	15
<b>Setting command line options</b>	15
Extended command line file	15
<b>Summary of assembler options</b>	16
<b>Descriptions of assembler options</b>	17
<b>Assembler operators</b>	29
<b>Precedence of operators</b>	29
<b>Summary of assembler operators</b>	29
Unary operators – 1	29
Multiplicative arithmetic operators – 2	30
Additive arithmetic operators – 3	30
Shift operators – 4	30
AND operators – 5	30
OR operators – 6	30
Comparison operators – 7	31
Operator synonyms	31
<b>Description of operators</b>	32
<b>Assembler directives</b>	43
<b>Summary of assembler directives</b>	43
<b>Module control directives</b>	47
Syntax	48
Parameters	48
Descriptions	48

<b>Symbol control directives .....</b>	50
Syntax .....	50
Parameters .....	50
Descriptions .....	51
Examples .....	51
<b>Mode control directives .....</b>	52
Syntax .....	52
Description .....	53
Examples .....	53
<b>Section control directives .....</b>	54
Syntax .....	54
Parameters .....	55
Descriptions .....	55
Examples .....	56
<b>Value assignment directives .....</b>	57
Syntax .....	57
Parameters .....	57
Descriptions .....	58
<b>Conditional assembly directives .....</b>	58
Syntax .....	59
Parameters .....	59
Descriptions .....	59
Examples .....	60
<b>Macro processing directives .....</b>	60
Syntax .....	61
Parameters .....	61
Descriptions .....	61
Examples .....	65
<b>Listing control directives .....</b>	68
Syntax .....	68
Parameters .....	69
Descriptions .....	69
Examples .....	70

<b>C-style preprocessor directives</b>	72
Syntax	73
Parameters	73
Descriptions	74
Examples	76
<b>Data definition or allocation directives</b>	77
Syntax	77
Parameters	78
Descriptions	78
Examples	78
<b>Assembler control directives</b>	79
Syntax	80
Parameters	80
Descriptions	80
Examples	81
<b>Call frame information directives</b>	82
Syntax	83
Parameters	84
Descriptions	85
Simple rules	89
CFI expressions	91
Example	93
<b>Assembler pseudo-instructions</b>	97
<b>Summary</b>	97
<b>Descriptions of pseudo-instructions</b>	98
<b>Assembler diagnostics</b>	107
<b>Message format</b>	107
<b>Severity levels</b>	107
Options for diagnostics	107
Assembly warning messages	107
Command line error messages	107
Assembly error messages	108
Assembly fatal error messages	108

Assembler internal error messages .....	108
<b>Migrating to the ARM IAR Assembler .....</b>	<b>109</b>
<b>Introduction .....</b>	<b>109</b>
Thumb code labels .....	109
<b>Alternative register names .....</b>	<b>110</b>
<b>Alternative mnemonics .....</b>	<b>111</b>
<b>Operator synonyms .....</b>	<b>112</b>
<b>Warning messages .....</b>	<b>113</b>
<b>Index .....</b>	<b>115</b>



# Tables

1: Typographic conventions used in this guide .....	xiii
2: Assembler environment variables .....	4
3: Assembler error return codes .....	5
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 .....	13
10: Assembler options summary .....	16
11: Conditional list (-c) .....	17
12: Controlling case sensitivity in user symbols (-s) .....	25
13: Disabling assembler warnings (-w) .....	26
14: Including cross-references in assembler list file (-x) .....	27
15: Operator synonyms .....	31
16: Assembler directives summary .....	43
17: Module control directives .....	47
18: Symbol control directives .....	50
19: Mode control directives .....	52
20: Section control directives .....	54
21: Value assignment directives .....	57
22: Conditional assembly directives .....	58
23: Macro processing directives .....	60
24: Listing control directives .....	68
25: C-style preprocessor directives .....	72
26: Data definition or allocation directives .....	77
27: Assembler control directives .....	79
28: Call frame information directives .....	82
29: Unary operators in CFI expressions .....	92
30: Binary operators in CFI expressions .....	92
31: Ternary operators in CFI expressions .....	93

32: Code sample with backtrace rows and columns .....	94
33: Pseudo-instructions .....	97
34: Alternative register names .....	110
35: Alternative mnemonics .....	111
36: Operator synonyms .....	112

# Preface

Welcome to the ARM® IAR Assembler Reference Guide. The purpose of this guide is to provide you with detailed reference information that can help you to use the ARM 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 ARM core and need to get detailed reference information on how to use the ARM® IAR Assembler. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the ARM core. Refer to the documentation from Advanced RISC Machines Ltd for information about the ARM core
- 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 ARM® IAR Assembler, you should read the chapter *Introduction to the ARM 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 *ARM® IAR Embedded Workbench® IDE User Guide*. They give product overviews, as well as tutorials that can help you get started. The *ARM® IAR C/C++ Development 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 ARM 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.
- *Assembler pseudo-instructions* lists the available pseudo-instructions and gives examples of their use.
- *Assembler diagnostics* contains information about the formats and severity levels of diagnostic messages.
- *Migrating to the ARM IAR Assembler* contains information that is useful when you want to use the ARM IAR Assembler with source code that was originally developed for another assembler.

---

## Other documentation

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

- Using the IAR Embedded Workbench® IDE with the IAR C-SPY® Debugger, refer to the *ARM® IAR Embedded Workbench® IDE User Guide*
- Programming for the ARM IAR C/C++ Compiler and using the IAR ILINK Linker, refer to the *ARM® IAR C/C++ Development Guide*
- Using the IAR DLIB Library, refer to the online help system
- Porting application code and projects created with a previous ARM IAR Embedded Workbench, refer to the *ARM® IAR Embedded Workbench® Migration Guide*.

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
<code>computer</code>	Text that you enter or that appears on the screen.
<i>parameter</i>	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.
<b>bold</b>	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>reference</i>	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 ARM 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.

Refer to Advanced RISC Machines Ltd's hardware documentation for syntax descriptions of the instruction mnemonics.

---

## 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 ARM core 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 ARM core. Refer to Advanced RISC Machines Ltd's 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 *ARM® IAR Embedded Workbench® IDE User Guide*
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the *ARM® IAR C/C++ Development Guide*
- In the IAR Embedded Workbench IDE, you can base a new project on a *template* for an assembler project.

---

## Modular programming

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

- efficient program development
- reuse of modules
- maintenance.

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

Typically, you write your assembler code in assembler source files; each file becomes a named *module*. By dividing your source code into many small source files, you will get many small modules. Each module can be further divided into different subroutines.

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

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

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

- Create many small modules, one per source file
- In each module, divide your assembler source code into small subroutines (corresponding to *functions* on the C level)
- Divide your assembler source code in *sections*, to gain more precise control of how your code and data finally will be placed in memory
- Collect your routines in libraries, which means that you can reduce the number of object files and make the modules conditionally linked.

---

## External interface details

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

You can use the assembler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *ARM® 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:

```
iasmarm [options] [sourcefile] [options]
```

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

```
iasmarm prog -r
```

By default, the ARM IAR Assembler recognizes the filename extensions `s`, `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 `o`.

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 `iasmarm` command; see *Assembler invocation syntax*, page 3.

- Via environment variables

The assembler automatically appends the value of the environment variables to every command line; see *Environment variables*, page 4.

- Via a text file by using the `-f` option; see *-f*, page 19.

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 `IASMARM` 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 ARM IAR Assembler:

Environment variable	Description
<code>IASMARM</code>	Specifies command line options; for example: <code>set IASMARM=-L -ws</code>
<code>IASMARM_INC</code>	Specifies directories to search for include files; for example: <code>set IASMARM_INC=c:\myinc\</code>

Table 2: Assembler environment variables

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

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

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

## ERROR RETURN CODES

When using the ARM 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 (only if the <code>-ws</code> option is used).
2	There were 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 <code>:</code> (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>	A list of operands, separated by commas.
<i>comment</i>	Comment, preceded by a <code>;</code> (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 ARM IAR Assembler supports the syntax for assembler instructions as described in the *ARM Architecture Reference Manual*. It complies with the requirement of the ARM architecture on word alignment. Any instructions in a code section placed on an odd address will result in an error on cores with word alignment restrictions.

## 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 *Assembler operators*, page 29.

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), . (period).

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

**Note:** It is not possible to have two symbols in one expression, or any other complex expression, unless the expression can be resolved at assembly time. In this case, the assembler will generate an error.

### 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, b'1010

Table 4: Integer constant formats

Integer type	Example
Octal	1234q, q'1234
Decimal	1234, -1, d'1234
Hexadecimal	0xFFFFh, 0xFFFF, h'FFFF

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 ARM IAR Assembler will accept floating-point values as constants and convert them into IEEE single-precision (signed 32-bit) floating-point format and double-precision (signed 64-bit), or fractional format.

Floating-point numbers can be written in the format:

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

The following table shows some valid examples:

Format	Value
10.23	$1.023 \times 10^1$

Table 6: Floating-point constants

Format	Value
1.23456E-24	$1.23456 \times 10^{-24}$
1.0E3	$1.0 \times 10^3$

Table 6: Floating-point constants (Continued)

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).

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

```
`strange#label`
```

Case is insignificant for built-in symbols like instructions, registers, operators, and directives. For user-defined symbols case is by default significant but can be turned on and off using the **Case sensitive user symbols** (-s) assembler option. See -s, page 25 for additional information.

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

Note that symbols and labels are byte addresses. For additional information, see *Generating a lookup table*, page 78.

## 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 `.` (period) sign. For example:

```
SECTION MYCODE : CODE (2)
CODE32
B . ; Loop forever
END
```

## REGISTER SYMBOLS

The following table shows the existing predefined register symbols:

Name	Size	Description
CPSR	32 bits	Current program status register
D0–D15	64 bits	Vector floating-point coprocessor registers for double precision
FPEXC	32 bits	Vector floating-point coprocessor, exception register
FPSCR	32 bits	Vector floating-point coprocessor, status and control register
FPSID	32 bits	Vector floating-point coprocessor, system ID register
R0–R12	32 bits	General purpose registers
R13 (SP)	32 bits	Stack pointer
R14 (LR)	32 bits	Link register
R15 (PC)	32 bits	Program counter
S0–S31	32 bits	Vector floating-point coprocessor registers for single precision
SPSR	32 bits	Saved process status register

Table 7: Predefined register symbols

In addition, specific cores may allow you to use other register symbols, for example APSR for the Cortex-M3, if required by the instruction syntax.

## PREDEFINED SYMBOLS

The ARM 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
<code>__ARMVFP__</code>	Identifies whether floating-point instructions for a vector floating-point coprocessor have been enabled or not, --fpu. Expands to the number 1 for VFPv1, and to the number 2 for VFPv2. If floating-point instructions are disabled (default), the symbol is undefined.
<code>__BUILD_NUMBER__</code>	A unique integer that identifies the build number of the assembler currently in use.
<code>__DATE__</code>	The current date in dd/Mmm/yyyy format (string).
<code>__FILE__</code>	The name of the current source file (string).
<code>__IAR_SYSTEMS_ASM__</code>	IAR assembler identifier (number).
<code>__IASMARM__</code>	An integer that is set to 1 when the code is assembled with the ARM IAR Assembler.
<code>__LINE__</code>	The current source line number (number).
<code>__LITTLE_ENDIAN__</code>	Identifies the byte order in use. Expands to the number 1 when the code is compiled with the little-endian byte order, and to the number 0 when big-endian code is generated. Little-endian is the default.
<code>__TID__</code>	Target identity, consisting of two bytes (number). The high byte is the target identity, which is 0x4F (=decimal 79) for the ARM IAR Assembler. The low byte is not used.
<code>__TIME__</code>	The current time in hh:mm:ss format (string).
<code>__VER__</code>	The version number in integer format; for example, version 4.17.5 is returned as 4017005 (number).

Table 8: Predefined symbols

In addition, predefined symbols are defined that allow you to identify the core you are assembling for, for example `__ARM5__` and `__CORE__`. For more details, see the *ARM® IAR C/C++ Development Guide*.

## 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:

```
EXTERN printstr
SECTION MYDATA : DATA (2)
DATA
tim    DC8 __TIME__ ; time string
SECTION MYCODE : CODE (2)
CODE32
ADR R0, tim ; load address of string
BL printstr ; routine to print string
```

## 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__ > 4016005) ; New assembler version
...
...
#
#else                      ; Old assembler version
...
...
#endif
```

See *Conditional assembly directives*, page 58.

## ABSOLUTE AND RELOCATABLE EXPRESSIONS

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

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

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

For example, a program could define the sections MYDATA and MYCODE as follows:

```
EXTERN third
SECTION MYDATA : DATA (2)
first: DC32 3
second: DC32 4
```

Then in the section MYCODE, the following relocatable expressions are legal:

```
SECTION MYCODE : CODE (2)
CODE32
; MYDATA must be linked in the range 0-255,
; otherwise the immediate values #first etc.
; will be out of range
MOV R1,#first
MOV R2,#second
MOV R3,#third
LDR R1,=first+4
LDR R2,=second
LDR R3,=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 (section offset) is not allowed.

## Fixed

The expression must be fixed, which means that it must not depend on variable-sized instructions. A variable-sized instruction is an instruction that 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 section. The notation is hexadecimal.
- The data field shows the data generated by the source line. The notation is hexadecimal. Unresolved values are represented by .... (periods), where two periods signify one byte. These unresolved values 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).

Table 9: Symbol and cross-reference table

Information	Description
Sections	The name of the section that this symbol is defined relative to.
Value/Offset	The value (address) of the symbol within the current module, relative to the beginning of the current section.

Table 9: Symbol and cross-reference table (Continued)

## Programming hints

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

### ACCESSING SPECIAL FUNCTION REGISTERS

Specific header files for a number of ARM devices are included in the IAR Systems product package, in the `\arm\inc` directory. These header files define the processor-specific special function registers (SFRs) and in some cases the interrupt vector numbers.

Because the header files are also intended to be used with the ARM IAR C/C++ Compiler, the SFR declarations are made with macros. The macros that convert the declaration to assembler or compiler syntax are defined in the `io_macros.h` file.

The header files are also suitable to use as templates, when creating new header files for other ARM derivatives.

#### Example

The USART write address `0xFFFFD0000` of the device is defined in the `ioat91m40400.h` file as:

```
__IO_REG32_BIT(__US_CR, 0xffffd0000, __WRITE, __usartcr_bits)
```

The declaration is converted by macros defined in the file `io_macros.h` to:

```
__US_CR DEFINE 0xffffd0000
```

### 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 79.

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

---

## Setting command line options

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

```
iasmarm [options] [sourcefile] [options]
```

These items must be separated by one or more spaces or tab characters.

If all the optional parameters are omitted the assembler will display a list of available options a screenful at a time. Press Enter to display the next screenful.

For example, when assembling the source file `power2.s`, use the following command to generate a list file to the default filename (`power2.lst`):

```
iasmarm power2 -L
```

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

```
iasmarm power2 -l list.lst
```

Some other options accept a string that is not a filename. This is included after the option letter, but without a space. For example, to generate a list file to the default filename but in the subdirectory named `list`:

```
iasmarm power2 -Llist\
```

**Note:** The subdirectory you specify must already exist. The trailing backslash is required to separate the name of the subdirectory and the default filename.

## EXTENDED COMMAND LINE FILE

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

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

```
iasmarm -f extend.xcl
```

## Summary of assembler options

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

Command line option	Description
<code>-B</code>	Macro execution information
<code>-C</code>	Conditional list
<code>--cpu</code>	Core configuration
<code>-D</code>	Defines a symbol
<code>-E</code>	Maximum number of errors
<code>-e</code>	Generates code in big-endian byte order
<code>--endian</code>	Specifies the byte order for code and data
<code>-f</code>	Extends the command line
<code>--fpu</code>	Selects the type of floating-point unit
<code>-G</code>	Opens standard input as source
<code>-I</code>	Includes paths
<code>-i</code>	Lists #included text
<code>-j</code>	Enables alternative register names, mnemonics, and operators
<code>-L</code>	Lists to prefixed source name
<code>-l</code>	Output list file
<code>-M</code>	Macro quote characters
<code>-N</code>	Omit header from assembler listing
<code>-n</code>	Enables support for multibyte characters
<code>-O</code>	Sets object filename prefix
<code>-o</code>	Sets object filename
<code>-p</code>	Lines/page
<code>-r</code>	Generates debug information
<code>-S</code>	Sets silent operation

Table 10: Assembler options summary

Command line option	Description
-s	Case sensitive user symbols
-t	Tab spacing
-U	Undefines a symbol
-w	Disables warnings or changes the exit code
-x	Includes cross-references

Table 10: Assembler options summary (Continued)

## Descriptions 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.

---

-B -B

Use this option to make the assembler print macro execution information to the standard output stream on every call of a macro. The information consists of:

- The name of the macro
- The definition of the macro
- The arguments to the macro
- The expanded text of the macro.

This option is mainly used in conjunction with the list file options **-L** or **-l**; for additional information, see page 21.



**Project>Options>Assembler >List>Macro execution info**

---

-c -c {DMEAO}

Use this option to control the contents of the assembler list file. This option is mainly used in conjunction with the list file options **-L** and **-l**; see page 21 for additional information.

The following table shows the available parameters:

Command line option	Description
-cD	Disable list file

Table 11: Conditional list (-c)

Command line option	Description
-cM	Macro definitions
-cE	No macro expansions
-cA	Assembled lines only
-cO	Multiline code

Table 11: Conditional list (-c) (Continued)



To set related options, select:

**Project>Options>Assembler >List**

---

--cpu --cpu *target\_core*

Use the --cpu option to specify the target core and get the correct instruction set.

Valid values for *target\_core* are values such as ARM7TDMI and architecture versions, for example 4T. ARM7TDMI is the default.**Project>Options>General Options >Target>Processor variant>Core**

---

-D -D*symbol*[=*value*]Use this option to define a preprocessor symbol 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

For example, you could 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 in the command line as follows:

Production version: iasmarm prog

Test version: iasmarm 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:

```
iasmarm prog -DFRAMERATE=3
```



#### **Project>Options>Assembler>Preprocessor>Defined symbols**

---

**-E** *-Enumber*

This option specifies the maximum number of errors that the assembler will report.

By default, the maximum number is 100. The `-E` option allows you to decrease or increase this number to see more or fewer errors in a single assembly.



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

---

**-e** *-e*

This option causes the assembler to generate code and data in big-endian byte order. The default byte order is little-endian.



#### **Project>Options>General Options >Target>Endian mode**

---

**--endian** *--endian={little|l|big|b}*

This option specifies the byte order of the generated code and data.



#### **Project>Options>General Options >Target>Endian mode**

---

**-f** *-f filename*

This option extends the command line with text read from the file named `extend.xcl`. 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.

#### **Example**

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

```
iasmarm prog -f extend.xcl
```



To set this option, use:

#### **Project>Options>Assembler>Extra Options**

---

--fpu --fpu={VFPv1 | VFPv2 | VFP9-S | none}

Use the --fpu option to specify the target floating-point coprocessor to get the correct instruction set.

The following parameters are available:

VFPv1	A vector floating-point unit conforming to the VFPv1 architecture, such as the VFP10 rev 0.
-------	---

VFPv2	A system that implements a VFP unit conforming to the VFPv2 architecture, such as the VFP10 rev 1.
-------	--

VFP9-S	VFP9-S, an implementation of the VFPv2 architecture that can be used with the ARM9E family of CPU cores. Selecting the VFP9-S coprocessor is therefore identical to selecting the VFPv2 architecture.
--------	---

none (default)	The software floating-point library is used.
----------------	--

The --fpu option is automatically set if you use the --cpu option to select a target core that has a floating-point unit.



### Project>Options>General Options >Target>FPU

---

-G -G

This option causes the assembler to read the source from the standard input stream, rather than from a specified source file.

When -G is used, no source filename may be specified.



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

---

-I -Iprefix

Use this option to specify paths to be used by the preprocessor by adding 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 `IASMARM_INC` environment variable. The -I option allows you to give the assembler the names of directories where it will also search if it fails to find the file in the current working directory.

### 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 finally in the directory `c:\thisproj\headers\`.

You can also specify the include path with the `IASMARM_INC` environment variable, see *Environment variables*, page 4.



### **Project>Options>Assembler >Preprocessor>Additional include directories**

---

`-i -i`

Lists `#include` files in the list file.

By default, the assembler does not list `#include` file lines since these often come from standard files and would waste space in the list file. The `-i` option allows you to list these file lines.



### **Project>Options>Assembler >List>#included text**

---

`-j -j`

Enables alternative register names, mnemonics, and operators in order to increase compatibility with other assemblers and allow porting of code.

For additional information, see *Operator synonyms*, page 31, and the chapter *Migrating to the ARM IAR Assembler*.



### **Project>Options>Assembler >Language>Allow alternative register names, mnemonics and operands**

---

`-L -L [prefix]`

By default the assembler does not generate a list file. Use this option to make the assembler generate one and sent it to file `[prefix] sourcename.lst`.

To simply generate a listing, use the `-L` option without a prefix. The listing is sent to the file with the same name as the source, but the extension will be `.lst`.

The `-L` option lets you specify a prefix, for example to direct the list file to a subdirectory. Notice that you cannot include a space before the prefix.

`-L` may not be used at the same time as `-l`.

### Example

To send the list file to `list\prog.lst` rather than the default `prog.lst`:

```
iasasmr prog -Llist\
```



To set related options, select:

**Project>Options>Assembler >List**

---

`-l -l filename`

Use this option to make the assembler generate a listing and send it to the file `filename`. If no extension is specified, `lst` is used. Notice that you must include a space before the filename.

By default, the assembler does not generate a list file. The `-l` option generates a listing, and directs it to a specific file. To generate a list file with the default filename, use the `-L` option instead.



To set related options, select:

**Project>Options>Assembler >List**

---

`-M -Mab`

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

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

### Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

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

```
iasmarm filename -M'<>'
```



#### Project>Options>Assembler >Language>Macro quote characters

-N -N

Use this option to omit the header section that is printed by default in the beginning of the list file.

This option is useful in conjunction with the list file options -L or -l; see page 21 for additional information.



#### Project>Options>Assembler >List>Include header

-n -n

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 C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.



#### Project>Options>Assembler >Language>Enable multibyte support

-O -Oprefix

Use this option to set the prefix to be used on the name of the object file. Notice that you cannot include a space before the prefix.

By default the prefix is null, so the object filename corresponds to the source filename (unless -O is used). The -O option lets you specify a prefix, for example to direct the object file to a subdirectory.

Notice that -O may not be used at the same time as -o.

#### **Example**

To send the object code to the file obj\prog.o rather than to the default file prog.o:

```
iasmarm prog -Oobj\
```



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

---

-o *filename*

This option sets the filename to be used for the object file. Notice that you must include a space before the filename. If no extension is specified, *o* is used.

The option -o may not be used at the same time as the option -O.

### **Example**

For example, the following command puts the object code to the file *obj.o* instead of the default *prog.o*:

```
iasmarm prog -o obj
```

Notice that you must include a space between the option itself and the filename.



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

---

-p *plines*

The -p option sets the number of lines per page to *lines*, which must be in the range 10 to 150.

This option is used in conjunction with the list options -L or -l; see page 21 for additional information.



**Project>Options>Assembler>List>Lines/page**

---

-r *-r*

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

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



**Project>Options>Assembler >Output>Generate debug information**

---

-S *-S*

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

By default, the assembler sends various insignificant messages via the standard output stream. Use the -S 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.

---

`-s -s{+|-}`

Use the `-s` option to control whether the assembler is sensitive to the case of user symbols:

Command line option	Description
<code>-s+</code>	Case sensitive user symbols
<code>-s-</code>	Case insensitive user symbols

Table 12: Controlling case sensitivity in user symbols (-s)

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



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

---

`-t -tn`

By default the assembler sets 8 character positions per tab stop. The `-t` option allows you to specify a tab spacing to `n`, which must be in the range 2 to 9.

This option is useful in conjunction with the list options `-L` or `-l`; see page 21 for additional information.



**Project>Options>Assembler>List>Tab spacing**

---

`-U -Usymbol`

Use the `-U` option to undefine the predefined symbol `symbol`.

By default, the assembler provides certain predefined symbols; see *Predefined symbols*, page 10. The `-U` option allows you to undefine such a predefined symbol to make its name available for your own use through a subsequent `-D` option or source definition.

### Example

To use the name of the predefined symbol `__TIME__` for your own purposes, you could undefine it with:

```
iasmarm prog -U __TIME__
```



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

---

-w `-w[string] [s]`

By default, the assembler displays a warning message when it detects an element of the source which is legal in a syntactical sense, but may contain a programming error; see *Assembler diagnostics*, page 107, for details.

Use this option to disable warnings. The `-w` option without a range disables all warnings. The `-w` option with a range performs the following:

Command line option	Description
<code>-w+</code>	Enables all warnings
<code>-w-</code>	Disables all warnings
<code>-w+n</code>	Enables just warning <i>n</i>
<code>-w-n</code>	Disables just warning <i>n</i>
<code>-w+m-n</code>	Enables warnings <i>m</i> to <i>n</i>
<code>-w-m-n</code>	Disables warnings <i>m</i> to <i>n</i>

Table 13: Disabling assembler warnings (-w)

Only one `-w` option may be used on the command line.

By default, the assembler generates exit code 0 for warnings. Use the `-ws` option to generate exit code 1 if a warning message is produced.

### Example

To disable just warning 0 (unreferenced label), use the following command:

```
iasmarm prog -w-0
```

To disable warnings 0 to 8, use the following command:

```
iasmarm prog -w-0-8
```



To set related options, select:

**Project>Options>Assembler>Diagnostics**

---

-x `-x{DI2}`

Use this option to make the assembler include a cross-reference table at the end of the list file.

This option is useful in conjunction with the list options `-L` or `-l`; see page 21 for additional information.

The following parameters are available:

Command line option	Description
-xD	#defines
-xI	Internal symbols
-x2	Dual line spacing

Table 14: Including cross-references in assembler list file (-x)

Project>Options>Assembler>List>Include cross reference





# 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 7 (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))`

---

## Summary of assembler operators

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

### UNARY OPERATORS – I

<code>+</code>	Unary plus.
<code>-</code>	Unary minus.
<code>!</code>	Logical NOT.
<code>~</code>	Bitwise NOT.
<code>LOW</code>	Low byte.
<code>HIGH</code>	High byte.

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

## MULTIPLICATIVE ARITHMETIC OPERATORS – 2

*	Multiplication.
/	Division.
%	Modulo.

## ADDITIVE ARITHMETIC OPERATORS – 3

+	Addition.
-	Subtraction.

## SHIFT OPERATORS – 4

>>	Logical shift right.
<<	Logical shift left.

## AND OPERATORS – 5

&&	Logical AND.
&	Bitwise AND.

## OR OPERATORS – 6

	Logical OR.
	Bitwise OR.
XOR	Logical exclusive OR.

`^` Bitwise exclusive OR.

## COMPARISON OPERATORS – 7

<code>=, ==</code>	Equal.
<code>&lt;&gt;, !=</code>	Not equal.
<code>&gt;</code>	Greater than.
<code>&lt;</code>	Less than.
<code>UGT</code>	Unsigned greater than.
<code>ULT</code>	Unsigned less than.
<code>&gt;=</code>	Greater than or equal.
<code>&lt;=</code>	Less than or equal.

## OPERATOR SYNONYMS

A number of operator synonyms have been defined for compatibility with other assemblers::

Operator synonym	Operator	Function
<code>:AND:</code>	<code>&amp;</code>	Bitwise AND
<code>:EOR:</code>	<code>^</code>	Bitwise exclusive OR
<code>:LAND:</code>	<code>&amp;&amp;</code>	Logical AND
<code>:LEOR:</code>	<code>XOR</code>	Logical exclusive OR
<code>:LNOT:</code>	<code>!</code>	Logical NOT
<code>:LOR:</code>	<code>  </code>	Logical OR
<code>:MOD:</code>	<code>%</code>	Modulo
<code>:NOT:</code>	<code>~</code>	Bitwise NOT
<code>:OR:</code>	<code> </code>	Bitwise OR
<code>:SHL:</code>	<code>&lt;&lt;</code>	Logical shift left
<code>:SHR:</code>	<code>&gt;&gt;</code>	Logical shift right

Table 15: Operator synonyms

**Note:** The operator synonyms are enabled by the option `-j`. The ARM operators and the operator synonyms may have different precedence. For more information about the precedence of the operators and their synonyms, see the following reference sections. See also the chapter *Migrating to the ARM IAR Assembler*.

## Description of operators

The following sections give detailed descriptions of each assembler operator. See *Expressions, operands, and operators*, page 6, for related information. The number within parentheses specifies the priority of the operator.

- 
- \* Multiplication (2).

\* 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**

$2 * 2 \rightarrow 4$   
 $-2 * 2 \rightarrow -4$

---

- + Unary plus (1).

Unary plus operator.

**Example**

$+3 \rightarrow 3$   
 $3 * +2 \rightarrow 6$

---

- + Addition (3).

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**

$92 + 19 \rightarrow 111$   
 $-2 + 2 \rightarrow 0$   
 $-2 + -2 \rightarrow -4$

---

- Unary minus (1).

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**

$-3 \rightarrow -3$   
 $3 * -2 \rightarrow -6$   
 $4 -- 5 \rightarrow 9$

---

- Subtraction (3).

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**

$92 - 19 \rightarrow 73$   
 $-2 - 2 \rightarrow -4$   
 $-2 -- 2 \rightarrow 0$

---

- / Division (2).

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

**Example**

$9 / 2 \rightarrow 4$   
 $-12 / 3 \rightarrow -4$   
 $9 / 2 * 6 \rightarrow 24$

---

- < 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**

$-1 < 2 \rightarrow 1$   
 $2 < 1 \rightarrow 0$   
 $2 < 2 \rightarrow 0$

---

<= Less than or equal (7)

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

**Example**

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

---

<>, != Not equal (7).

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

**Example**

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

---

=, == Equal (7).

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

**Example**

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

---

> Greater than (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
```

---

$\geq$  Greater than or equal (7).

$\geq$  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
```

---

$\&&$  (:LAND:) Logical AND (5).

Use  $\&&$  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).

**Note:** The precedence of :LAND: is 8.

**Example**

```
B'1010 && B'0011 → 1
B'1010 && B'0101 → 1
B'1010 && B'0000 → 0
```

---

$\&$  (:AND:) Bitwise AND (5).

Use  $\&$  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.

**Note:** The precedence of :AND: is 3.

**Example**

```
B'1010 & B'0011 → B'0010
B'1010 & B'0101 → B'0000
B'1010 & B'0000 → B'0000
```

---

$\sim$  (:NOT:) Bitwise NOT (1).

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

**Example**

```
~ B'1010 → B'1111111111111111111111110101
```

---

| (:OR:) Bitwise OR (6).

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

**Note:** The precedence of `:OR:` is 3.

**Example**

```
B'1010 | B'0101 → B'1111
B'1010 | B'0000 → B'1010
```

---

## ^ (:EOR:) Bitwise exclusive OR (6).

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

**Note:** The precedence of `:EOR:` is 3.

**Example**

```
B'1010 ^ B'0101 → B'1111
B'1010 ^ B'0011 → B'1001
```

---

## % (:MOD:) Modulo (2).

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

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

**Example**

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

---

## ! (:LNNOT:) Logical NOT (1).

Use `!` to negate a logical argument.

**Example**

```
! B'0101 → 0
! B'0000 → 1
```

---

|| (:LOR:) Logical OR (6).

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

**Example**

B'1010 || B'0000 → 1  
B'0000 || B'0000 → 0

---

BYTE1 First byte (1).

BYTE1 takes a single operand, which is interpreted as an unsigned 32-bit integer value. The result is the unsigned, 8-bit integer value of the lower order byte of the operand.

**Example**

BYTE1 0xABCD → 0xCD

---

BYTE2 Second byte (1).

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

**Example**

BYTE2 0x12345678 → 0x56

---

BYTE3 Third byte (1).

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 (1).

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

**Example**

BYTE4 0x12345678 → 0x12

---

**DATE** Current time/date (1).

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
```

---

**HIGH** High byte (1).

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 (1).

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

### **Example**

```
HWRD 0x12345678 → 0x1234
```

---

LOW Low byte (1).

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

**Example**

LOW 0xABCD → 0xCD

---

LWRD Low word (1).

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

**Example**

LWRD 0x12345678 → 0x5678

---

SFB Section begin (1).

**Syntax**

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

**Parameters**

*section*                   The name of a section, which must be defined before SFB is used.

*offset*                   An optional offset from the start address. The parentheses are optional if *offset* is omitted.

**Description**

SFB accepts a single operand to its right. The operand must be the name of a section.

The operator evaluates to the absolute address of the first byte of that section. This evaluation takes place at linking time.

**Example**

```
NAME demo
SECTION MYCODE : CODE (2)
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 section.

---

SFE Section end (1).

### Syntax

SFE (*section* [ $\{ + \mid - \}$  *offset*])

### Parameters

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

### Description

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

### Example

```
NAME demo
SECTION MYCODE : CODE (2)
...
SECTION MYCONST : CONST (2)
end: DC32 SFE (MYCODE)
END
```

Even if the above code is linked with many other modules, end will still be set to the address of the last byte of the section.

The size of the section MYCODE can be calculated as:

SFE (MYCODE) – SFB (MYCODE)

---

<< (:SHL:) Logical shift left (4).

Use << 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

```
B'00011100 << 3 → B'11100000
B'0000011111111111 << 5 → B'11111111111100000
14 << 1 → 28
```

---

>> (:SHR:) Logical shift right (4).

Use >> 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.

**Note:** The precedence of :SHR: is 2.5.

**Example**

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

---

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 the 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
```

---

XOR (:LEOR:) Logical exclusive OR (6).

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.

**Note:** The precedence of :LEOR: is 8.

**Example**

```
B'0101 XOR B'1010 → 0
B'0101 XOR B'0000 → 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 47
- *Symbol control directives*, page 50
- *Mode control directives*, page 52
- *Section control directives*, page 54
- *Value assignment directives*, page 57
- *Conditional assembly directives*, page 58
- *Macro processing directives*, page 60
- *Listing control directives*, page 68
- *C-style preprocessor directives*, page 72
- *Data definition or allocation directives*, page 77
- *Assembler control directives*, page 79
- *Call frame information directives*, page 82.

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

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

Table 16: Assembler directives summary

Directive	Description	Section
#ifndef	Assembles instructions if a symbol is undefined.	C-style preprocessor
#include	Includes a file.	C-style preprocessor
#line	Changes the line numbers.	C-style preprocessor
#message	Generates a message on standard output.	C-style preprocessor
#pragma	Recognized but ignored.	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
AAPCS	Sets module attributes.	Module control
ALIAS	Assigns a permanent value local to a module.	Value assignment
ALIGNRAM	Aligns the program location counter by incrementing it.	Section control
ALIGNROM	Aligns the program location counter by inserting zero-filled bytes.	Section control
ARM	Interprets subsequent instructions as 32-bit (ARM) instructions.	Mode control
ASSIGN	Assigns a temporary value.	Value assignment
CASEOFF	Disables case sensitivity.	Assembler control
CASEON	Enables case sensitivity.	Assembler control
CFI	Specifies call frame information.	Call frame information
CODE16	Interprets subsequent instructions as 16-bit (Thumb) instructions.	Mode control
CODE32	Interprets subsequent instructions as 32-bit (ARM) instructions.	Mode control
COL	Sets the number of columns per page.	Listing control
DATA	Defines an area of data within a code section.	Mode control
DC8	Generates 8-bit constants, including strings.	Data definition or allocation
DC16	Generates 16-bit constants.	Data definition or allocation
DC24	Generates 24-bit constants.	Data definition or allocation

Table 16: Assembler directives summary (Continued)

Directive	Description	Section
DC32	Generates 32-bit constants.	Data definition or allocation
DCB	Generates 8-bit byte constants, including strings.	Data definition or allocation
DCD	Generates 32-bit long word constants.	Data definition or allocation
DCW	Generates 16-bit word constants, including strings.	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
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
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
ENDR	Ends a repeat structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Section control
EXITM	Exits prematurely from a macro.	Macro processing
EXTERN	Imports an external symbol.	Symbol control
EXTRN	Imports an external symbol.	Symbol control
EXTWEAK	Imports an external symbol; the symbol may be undefined.	Symbol control

Table 16: Assembler directives summary (Continued)

Directive	Description	Section
IF	Assembles instructions if a condition is true.	Conditional assembly
IMPORT	Imports an external symbol.	Symbol control
INCLUDE	Includes a file.	Assembler control
LIBRARY	Begins a module; an alias for PROGRAM and NAME.	Module control
LOCAL	Creates symbols local to a macro.	Macro processing
LSTCND	Controls conditional assembler listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembler-listing output.	Listing control
LSTPAG	Retained for backward compatibility reasons. Recognized but ignored.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
LSTXRF	Generates a cross-reference table.	Listing control
LTORG	Directs the current literal pool to be assembled immediately following the directive.	Assembler control
MACRO	Defines a macro.	Macro processing
MODULE	Begins a module; an alias for PROGRAM and NAME.	Module control
NAME	Begins a program module.	Module control
ODD	Aligns the program location counter to an odd address.	Section control
OVERLAY	Recognized but ignored.	Symbol control
PAGE	Retained for backward compatibility reasons.	Listing control
PAGSIZ	Retained for backward compatibility reasons.	Listing control
PRESERVE8	Sets a module attribute.	Module control
PROGRAM	Begins a module.	Module control
PUBLIC	Exports symbols to other modules.	Symbol control
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.	Symbol control
RADIX	Sets the default base.	Assembler control
REPT	Assembles instructions a specified number of times.	Macro processing
REPTC	Repeats and substitutes characters.	Macro processing

Table 16: Assembler directives summary (Continued)

Directive	Description	Section
REPTI	Repeats and substitutes strings.	Macro processing
REQUIRE	Forces a symbol to be referenced.	Symbol control
REQUIRE8	Sets a module attribute.	Module control
RSEG	Begins a section.	Section control
RTMODEL	Declares runtime model attributes.	Module control
SECTION	Begins a section.	Section control
SECTION_TYPE	Sets ELF type and flags for a section.	Section control
SET	Assigns a temporary value.	Value assignment
SETA	Assigns a temporary value.	Value assignment
THUMB	Interprets subsequent instructions as Thumb execution-mode instructions.	Mode control
VAR	Assigns a temporary value.	Value assignment

Table 16: Assembler directives summary (Continued)

## Module control directives

Module control directives are used for marking the beginning and end of source program modules, and for assigning names to them. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Description	Expression restrictions
AAPCS	Sets module attributes that informs the linker that all exported functions in the module follows AAPCS.	The assembler does not verify that the claims are fulfilled.
END	Terminates the assembly of the last module in a file.	Locally defined symbols plus offset or integer constants
NAME	Begins a module.	No external references Absolute
PRESERVE8	Sets a module attribute that informs the linker that all exported functions in the module preserves an 8-byte aligned stack.	The assembler does not verify that the claims are fulfilled.
PROGRAM	Begins a module.	No external references Absolute
REQUIRE8	Sets a module attribute that informs the linker that the module requires an 8-byte aligned stack.	

Table 17: Module control directives

Directive	Description	Expression restrictions
RTMODEL	Declares runtime model attributes.	Not applicable

Table 17: Module control directives (Continued)

## SYNTAX

```
AAPCS [modifier [...]]  
END  
NAME symbol  
PRESERVE8  
PROGRAM symbol  
REQUIRE8  
RTMODEL key, value
```

## PARAMETERS

<i>key</i>	A text string specifying the key.
<i>modifier</i>	An AAPCS extension; possible values are INTERWORK and VFP, and they may be combined.
<i>symbol</i>	Name assigned to module.
<i>value</i>	A text string specifying the value.

## DESCRIPTIONS

### Beginning a module

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

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

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

### Terminating the source file

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

## Setting module attributes for AEABI compliance

You can set specific attributes on a module to inform the linker that the exported functions in the module are compliant to certain parts of the AEABI standard.

Use AAPCS to indicate that a module is compliant with the AAPCS specification. Use PRESERVE8 if the module preserves an 8-byte aligned stack and REQUIRE8 if an 8-byte aligned stack is expected.

Note that it is up to you to verify that the module in fact is compliant to these parts as the assembler does not verify this.

## 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 *ARM® IAR C/C++ Development Guide*.

### Examples

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

- MOD\_1 and MOD\_2 *cannot* be linked together since they have different values for runtime model `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.

Assembler source file f1.s :

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

Assembler source file f2.s :

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

Assembler source file f3.s :

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

## Symbol control directives

These directives control how symbols are shared between modules.

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

Table 18: Symbol control directives

### SYNTAX

```
EXTERN symbol [, symbol] ...
EXTWEAK symbol [, symbol] ...
IMPORT symbol [, symbol] ...
PUBLIC symbol [, symbol] ...
PUBWEAK symbol [, symbol] ...
REQUIRE symbol
```

### PARAMETERS

<i>label</i>	Label to be used as an alias for a C/C++ symbol.
<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 `ILINK`. If a module containing a `PUBLIC` definition of a symbol is linked with one or more modules containing `PUBWEAK` definitions of the same symbol, `ILINK` will use the `PUBLIC` definition.

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

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

## EXAMPLES

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

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

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

```
MODULE error
EXTERN print
PUBLIC err
SECTION MYCODE : CODE (2)
CODE16
PUSH {LR}
err ADR R0,msg
BL print
POP {PC}
SECTION MYDATA : DATA (2)
DATA
msg DC8 "***Error ***"
END
```

## Mode control directives

These directives provide control over the processor mode:

Directive	Description
ARM, CODE32	Subsequent instructions are assembled as 32-bit (ARM) instructions. Labels within a CODE32 area have bit 0 set to 0. Force 4-byte alignment.
CODE16	Subsequent instructions are assembled as 16-bit (Thumb) instructions, using the traditional CODE16 syntax. Labels within a CODE16 area have bit 0 set to 1. Force 2-byte alignment.
DATA	Defines an area of data within a code section, where labels work as in a CODE32 area.
THUMB	Subsequent instructions are assembled either as 16-bit Thumb instructions, or as 32-bit Thumb-2 instructions if the specified core supports the Thumb-2 instruction set. The assembler syntax follows the Unified Assembler syntax as specified by Advanced RISC Machines Ltd.

Table 19: Mode control directives

## SYNTAX

```
ARM
CODE16
CODE32
DATA
THUMB
```

## DESCRIPTION

To change between the Thumb and ARM processor modes, use the CODE16/THUMB and CODE32/ARM directives with the BX instruction (Branch and Exchange) or some other instruction that changes the execution mode. The CODE16/THUMB and CODE32/ARM mode directives do not assemble to instructions that change the mode, they only instruct the assembler how to interpret the following instructions.

Always use the DATA directive when defining data in a Thumb code section with DC8, DC16, or DC32, otherwise labels on the data will have bit 0 set.

**Note:** Be careful when porting assembler source code written for other assemblers. The IAR Assembler always sets bit 0 on Thumb code labels (local, external or public). See the chapter *Migrating to the ARM IAR Assembler* for details.

The assembler will initially be in CODE32/ARM mode, except if you specified a core which does not support ARM mode. In this case, the assembler will initially be in THUMB mode.

## EXAMPLES

### Changing the processor mode

The following example shows how a THUMB entry to an ARM function may be implemented:

```
MODULE example
SECTION MYCODE : CODE (2)

        THUMB
thumbEntryToFunction
        BX PC ; Branch to armEntryToFunction, and change execution
               ; mode.
        NOP ; For alignment only.
        ARM
armEntryToFunction
.
.
END
```

### Using the DATA directive

The following example shows how 32-bit labels are initialized after the DATA directive. The labels can be used within a CODE16 section.

```
MODULE example
SECTION MYCODE : CODE (2)
```

```

        CODE16 ; Thumb instruction set used
my_code_label1 ldr r0,my_data_label1
my_code_label2 nop
        DATA ; Remember the data directive,
; so that bit 0 is not set
; on labels
my_data_label1 DC32 0x12345678
my_data_label2 DC32 0x12345678

END

```

---

## Section control directives

The section 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
ALIGNRAM	Aligns the program location counter by incrementing it.	No external references Absolute
ALIGNROM	Aligns the program location counter by inserting zero-filled bytes.	No external references Absolute
EVEN	Aligns the program counter to an even address.	No external references Absolute
ODD	Aligns the program counter to an odd address.	No external references Absolute
RSEG	Begins a section; alias to SECTION.	No external references Absolute
SECTION	Begins an ELF section.	No external references Absolute

SECTION\_TYPE Sets ELF type and flags for a section.

---

Table 20: Section control directives

### SYNTAX

```

ALIGNRAM align
ALIGNROM align [,value]
EVEN [value]
ODD [value]
RSEG section :type [flag] [(align)]
SECTION section :type [flag] [(align)]

```

```
SECTION_TYPE type-expr {flags-expr}
```

## PARAMETERS

<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 0, except for code sections where the default is 1.
<i>flag</i>	NOROOT, ROOT NOROOT means that the section fragment is discarded by the linker if no symbols in this section fragment are referred to. Normally, all section fragments except startup code and interrupt vectors should set this flag. The default mode is ROOT which indicates that the section fragment must not be discarded.
	REORDER, NOREORDER REORDER starts a new section with the given name. The default mode is NOREORDER which starts a new fragment in the section with the given name, or a new section if no such section exists.
<i>section</i>	The name of the section.
<i>type</i>	The memory type, which can be either CODE, CONST, or DATA.
<i>value</i>	Byte value used for padding, default is zero.
<i>type-expr</i>	A constant expression identifying the ELF type of the section.
<i>flags-expr</i>	A constant expression identifying the ELF flags of the section.

## DESCRIPTIONS

### Beginning a relocatable section

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

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

To set the ELF type, and possibly the ELF flags for the newly created section, use SECTION\_TYPE. By default, the values of the flags will be zero. For information about valid values, refer to the ELF documentation.

## Aligning a section

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

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

ALIGNROM 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 ALIGNROM 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 to a specified address boundary. The expression gives the power of two to which the program location counter should be aligned. ALIGNRAM aligns by incrementing the program location counter; no data is generated.

## EXAMPLES

### Beginning a relocatable section

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

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

```

EXTERN subrtn, divrtn
SECTION MYDATA : DATA (2)

        DATA
functable:
f1:      DC32 subrtn
          DC32 divrtn

SECTION MYCODE : CODE (2)
CODE32
main:
        LDR R0,=f1 ; get address
        LDR PC,[R0] ; jump to it

```

## Aligning a section

This example shows how ALIGNRAM is used.

```

NAME align
SECTION MYDATA : DATA (6) ; Start a relocatable data
; section and verify that it
; is correctly aligned to a 64-byte boundary.
DATA
target1 DS16 1 ; Two bytes of data
ALIGNRAM 6 ; Align to a 64-byte boundary
results DS8 64 ; Create a 64-byte table
target2 DS16 1 ; Two bytes of data
ALIGNRAM 3 ; Align to an 8-byte boundary
ages DS8 64 ; Create another 64-byte table
END

```

## Value assignment directives

These directives are used for assigning values to symbols.

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

Table 21: Value assignment directives

### SYNTAX

```

label = expr
label ALIAS expr
label ASSIGN expr
label DEFINE expr
label EQU expr
label SET expr
label SETA expr
label VAR expr

```

### PARAMETERS

<i>expr</i>	Value assigned to symbol or value to be tested.
-------------	---

*label*                    Symbol to be defined.

## DESCRIPTIONS

### Defining a temporary value

Use ASSIGN, SET, SETA, 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. After the DEFINE directive, the symbol is known.

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.

## 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.	

Table 22: Conditional assembly directives

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

Table 22: Conditional assembly directives (Continued)

## 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.
<i>string1</i> = <i>string2</i>	The condition is true if <i>string1</i> and <i>string2</i> have the same length and contents.
<i>string1</i> <> <i>string2</i>	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 defines two different implementations of an addition depending on the number of arguments used in the macro:

```
SECTION MYCODE : CODE (2)
CODE32
?add MACRO a,b,c
      IF _args <3
          ADD a,a, #b
      ELSE
          ADD a,b, #c
      ENDIF
ENDM
```

If two macro-arguments are supplied then the first and second argument of the add instruction are assumed to be the same:

```
main:
    MOV R1, #0xF0
?add R1, 0xFF ;this
?add R1, R1, 0xFF ;and this
add R1, R1, #0xFF ;are the same as this
```

## 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.	

Table 23: Macro processing directives

Directive	Description	Expression restrictions
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 23: Macro processing directives (Continued)

## 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:

```
EXTERN errfunc
errmac MACRO text
    BL errfunc
    DATA
    DC8 text,0
ENDM
```

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

```
SECTION MYCODE : CODE (2)
CODE32
errmac 'Disk not ready'
```

The assembler will expand this to:

```
BL errfunc
DATA
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:

```
EXTERN errfunc
errmac MACRO
    BL      errfunc
    DATA
    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:

```
cmp_reg MACRO op
    CMP op
ENDM
```

The macro can be called using the macro quote characters:

```
SECTION MYCODE : CODE (2)
CODE32
cmp_reg <R3, R4>
```

You can redefine the macro quote characters with the -M command line option; see *-M*, page 22.

### 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

SECTION MYCODE : CODE (2)
    FILL 3, 4
    FILL 3
```

It generates the following code:

<pre>12 13 13.1 13.2 13.3</pre>	<pre>SECTION MYCODE : CODE (2) FILL 3, 4 IF _args == 2 REPT 3 DC8 4</pre>
---------------------------------	---

```

13.4
13    00000000 04      ENDR
13    00000001 04      DC8 4
13    00000002 04      DC8 4
13.1
13.2
13.3
13.4
14
14.1
14.2
14.3
14.4
14.5
14    00000003 03      ENDIF
14.1
14.2

```

## 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. Include-file references `$file` are recorded and will be included during macro *expansion*.
- 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:

```
IO_PORT EQU 0x0100
        DATA
        SECTION MYDATA_A : DATA (2)
start: DC32 main           ; reset vector
        SECTION MYDATA_B : DATA (2)
        DATA
buffer DS8 512            ; Reserve a buffer of 512 bytes
        SECTION MYCODE : CODE (2)
        CODE32
main:  BL play
done:  B done

play:
        LDR R1,=buffer      ; Use R1 as pointer into buffer
        LDR R2,=IO_PORT     ; Use R2 as pointer to the port
        LDR R3,=512          ; Buffersize in R3
        ADD R3,R3,R1        ; R3 = one past the buffer
loop:
        LDRB R4,[R1],#1     ; Get byte data in R4,
        ; increment pointer into buffer
        STRB R4,[R2]         ; Write to the address pointed to by R2
        CMP R1, R3           ; Compare R1 to the address
        ; one past the buffer
        BNE loop             ; NOT EQUAL --> repeat
        MOV PC,LR             ; return
```

For efficiency we can recode this using a macro:

```
play:  MACRO buf, size, port
      LOCAL loop
```

```

        LDR R1,=buf      ; Use R1 as pointer into buffer
        LDR R2,=port     ; Use R2 as pointer to the port
        LDR R3,=size     ; Buffersize in R3
        ADD R3,R3,R1    ; R3 = one past the buffer
loop:
        LDRB R4,[R1],#1  ; Get byte data in R4
                           ; increment pointer into buffer
        STRB R4,[R2]     ; Write to the address
                           ; pointed to by R2
        CMP R1, R3      ; Compare R1 to the
                           ; address one past the buffer
        BNE loop        ; NOT EQUAL -->repeat
        ENDM
IO_PORT EQU 0x0100
SECTION RESET : CODE (2)
start: B main
SECTION MYDATA : DATA (2)
DATA
buffer: DS8 512          ; Reserve a buffer of 512 bytes
SECTION MYCODE : CODE (2)
CODE32
main:
        play buffer,512,IO_PORT
done:  B done

```

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

## Using REPTC and REPTI

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

```

NAME signon
EXTERN plotc
SECTION MYCODE : CODE (2)
CODE32

banner REPTC chr, "Welcome"
        MOV R0,#'chr'      ; Pass char in R0 as parameter
        BL plotc
        ENDR

```

This produces the following code:

```

1
2
3
NAME signon
EXTERN plotc

```

```

4           SECTION MYCODE : CODE (2)
5           CODE32
6           banner REPTC chr, "Welcome"
7           MOV R0,#'chr' ; Pass char in R0 as
8           ; parameter
9           BL plotc
10          ENDR
10.1        00000000 5700A0E3
10.2
10.3        00000004 .....
10.4        00000008 6500A0E3
10.5
10.6        0000000C .....
10.7        00000010 6C00A0E3
10.8
10.9        00000014 .....
10.10       00000018 6300A0E3
10.11
10.12       0000001C .....
10.13       00000020 6F00A0E3
10.14
10.15       00000024 .....
10.16       00000028 6D00A0E3
10.17
10.18       0000002C .....
10.19       00000030 6500A0E3
10.20
10.21       00000034 .....

```

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

```

NAME repti
EXTERN base, count, init
SECTION MYCODE : CODE (2)
CODE32
MOV R0,#0

banner REPTI addr, base, count, init
LDR R1,=addr
STR R0,[R1]
ENDR

```

This produces the following code:

```

1           NAME repti
2           EXTERN base, count, init
3           SECTION MYCODE : CODE (2)
4           CODE32
5           MOV R0,#0
00000000 0000A0E3

```

```

6                                banner REPTI addr, base, count, init
7                                LDR R1,=addr
8                                STR R0,[R1]
9                                ENDR
9.1   00000004 10109FE5      LDR R1,= base
9.2   00000008 000081E5      STR R0,[R1]
9.3   0000000C 0C109FE5      LDR R1,= count
9.4   00000010 000081E5      STR R0,[R1]
9.5   00000014 08109FE5      LDR R1,= init
9.6   00000018 000081E5      STR R0,[R1]

```

## Listing control directives

These directives provide control over the assembler list file.

Directive	Description
COL	Sets the number of columns per page.
LSTCND	Controls conditional assembly listing.
LSTCOD	Controls multi-line code listing.
LSTEXP	Controls the listing of macro-generated lines.
LSTMAC	Controls the listing of macro definitions.
LSTOUT	Controls assembly-listing output.
LSTPAG	Controls the formatting of output into pages.
LSTREP	Controls the listing of lines generated by repeat directives.
LSTXRF	Generates a cross-reference table.
PAGE	Generates a new page.
PAGSIZ	Sets the number of lines per page.

Table 24: Listing control directives

## SYNTAX

```

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

```

```
PAGE  
PAGESIZ lines
```

## PARAMETERS

*columns* An absolute expression in the range 80 to 132, default is 80  
*lines* An absolute expression in the range 10 to 150, default is 44

## 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 restrict the listing of output code to just the first line of code for a source line.

The default setting is LSTCOD+, which lists more than one line of code for a source line, if needed; that is, long ASCII strings will produce several lines of output. Code generation is *not* affected.

### 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.

## Specifying the list file format

Use `COL` to set the number of columns per page of the assembler list. The default number of columns is 80.

Use `PAGSIZ` to set the number of printed lines per page of the assembler list. The default number of lines per page is 44.

Use `LSTPAG+` to format the assembler output list into pages.

The default is `LSTPAG-`, which gives a continuous listing.

Use `PAGE` to generate a new page in the assembler list file if paging is active.

## 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
SECTION PROM : CODE (2)
CODE32

debug SET 0
begin IF debug
      BL print
      ENDIF
      LSTCND+
begin2 IF debug
      BL print
```

```

NOP
ENDIF

```

This will generate the following listing:

```

1           NAME lstcndtst
2           EXTERN print
3           SECTION PROM : CODE (2)
4           CODE32
5           debug SET 0
6           begin IF debug
7               BL print
8           ENDIF
9           LSTCND+
10          begin2 IF debug
13          ENDIF

```

The following example shows the effect of LSTCOD- on the generated code by a DATA directive:

```

NAME lstcodtst
SECTION MYDATA : DATA (2)
DATA
table1: DC32 1,10,100,1000,10000
LSTCOD-
table2:
DC32 1,10,100,1000,10000
END

```

This will generate the following listing:

```

1           NAME lstcodtst
2           SECTION MYDATA : DATA (2)
3           DATA
4   00000000 01000000A00 table1: DC32 1,10,100,1000,10000
      000064000000
      E80300001027
      0000
5           LSTCOD-
6           table2:
7   00000014 01000000A00* DC32 1,10,100,1000,10000
      END

```

### Controlling the listing of macros

The following example shows the effect of LSTMAC and LSTEXP:

```

SECTION MYCODE : CODE (2)
CODE32
dec2    MACRO arg

```

```

        ADD R1,R1,#-arg
        ADD R1,R1,#-arg
        ENDM
        LSTMAC-
inc2    MACRO arg
        ADD R1,R1,#arg
        ADD R1,R1,#arg
        ENDM

begin:
        dec2 127
LSTEXP-
        inc2 0x20

```

This will produce the following output:

```

1
2                      SECTION MYCODE : CODE (2)
3                      CODE32
4                      dec2 MACRO arg
5                         ADD R1,R1,#-arg
6                         ADD R1,R1,#-arg
7                         ENDM
8                         LSTMAC-
13                     begin:
14                     dec2 127
14.1   00000000 7F1041E2      ADD R1,R1,#-127
14.2   00000004 7F1041E2      ADD R1,R1,#-127
14.3
15                     ENDM
16                     LSTEXP-
                     inc2 0x20

```

## 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.

Table 25: C-style preprocessor directives

Directive	Description
#ifdef	Assembles instructions if a preprocessor symbol is defined.
#ifndef	Assembles instructions if a preprocessor symbol is undefined.
#include	Includes a file.
#line	Changes the line numbers of the source code lines immediately following the #line directive, or the filename of the file being compiled.
#message	Generates a message on standard output.
#pragma	This directive is recognized but ignored.
#undef	Undefines a preprocessor symbol.

Table 25: C-style preprocessor directives (Continued)

## SYNTAX

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

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.

`#include "filename"` searches the following directories in the specified order:

- 1 The source file directory.
- 2 The directories specified by the `-I` option, or options.
- 3 The current directory.

`#include <filename>` searches the following directories in the specified order:

- 1 The directories specified by the `-I` option, or options.
- 2 The current directory.

## Displaying errors

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

## Ignoring #pragma

A `#pragma` line is ignored by the assembler, making it easier to have header files common to C and assembler.

## 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 */

DC32 five,11,12
; Expands to "DC32 5          ; this comment is not OK"

DC32 six + seven,11,12      ; OK
; Expanded to "DC32 6+7,11,12"
```

## EXAMPLES

### Using conditional preprocessor directives

The following example uses `#ifdef` to check that a certain symbol is defined and in that case use two internally defined symbols. Otherwise, the same symbols are declared `EXTERN` and a message displayed by `#message`. The `STAND_ALONE` symbol can, for example, be defined on the command line via the `-D` option, see `-D`, page 18.

```

PROGRAM target
SECTION MYCODE : CODE (2)
CODE32
PUBLIC main
#ifndef STAND_ALONE
alpha EQU 0x20
beta EQU 0x22
#else

EXTERN alpha, beta
#message "Program depends on additional link information"
#endif

main:
    MOV R1,#alpha
    MOV R2,#beta
    ADD R2,R2,R1
    EOR R1,R1,R2 ; R1 = (alpha XOR (alpha + beta))

```

### 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.s`:

```

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

```

The macro definitions can then be included by use of `#include`:

```

NAME include
SECTION MYCODE : CODE (2)
CODE32
; standard macro definitions
#include "macros.s"

```

```
; program
main:
    xch R0,R1,R2
```

## Data definition or allocation directives

These directives define values or reserve memory. The column *Alias* in the following table shows the Advanced RISC Machines Ltd directive that corresponds to the IAR Systems directive. See *Expression restrictions*, page 12, for a description of the restrictions that apply when using a directive in an expression.

Directive	Alias	Description
DC8	DCB	Generates 8-bit constants, including strings.
DC16	DCW	Generates 16-bit constants.
DC24		Generates 24-bit constants.
DC32	DCD	Generates 32-bit constants.
DF32		Generates 32-bit floating-point constants.
DF64		Generates 64-bit floating-point constants.
DS8	DS	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 26: Data definition or allocation directives

### SYNTAX

```
DC8 expr [,expr] ...
DC16 expr [,expr] ...
DC24 expr [,expr] ...
DC32 expr [,expr] ...
DCB expr [,expr] ...
DCD expr [,expr] ...
DCW expr [,expr] ...
DF32 value [,value] ...
DF64 value [,value] ...
DS8 count
DS16 count
DS24 count
DS32 count
```

## PARAMETERS

<i>count</i>	A valid absolute expression specifying the number of elements to be reserved.
<i>expr</i>	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.*
<i>value</i>	A valid absolute expression or floating-point constant.

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

## DESCRIPTIONS

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

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

## EXAMPLES

### Generating a lookup table

The following example generates a lookup table of addresses to routines:

```

ADD_SELECTOR DEFINE 0
SUB_SELECTOR DEFINE 4
DIV_SELECTOR DEFINE 8
MUL_SELECTOR DEFINE 12
    NAME table
    SECTION VECTORS : CODE (2)
    CODE32
    EXTERN add_f, sub_f, div_f, mul_f
start   B main           ; RESET vector
        SECTION MYDATA : DATA (2)
        DATA
                ; pointer table to floating point routines

table:  DC32 add_f, sub_f, div_f, mul_f
PI:     DC32 3.1415927
radius: DC32 1.3e2
        SECTION MYCODE : CODE (2)
        CODE32
main:
        LDR R1, =PI      ; Read up address to P1
        LDR R2, =radius   ; Do the same with radius...
        LDR R0, =table    ; Put base address to table in R0

```

```

        LDR R0, [R0,#MUL_SELECTOR] ; Read address to mul_f from
table
        MOV PC,R0 ; goto mul_f

```

## Defining strings

To define a string:

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

To define a string which includes a trailing zero:

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

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

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

## Reserving space

To reserve space for 10 bytes:

```
table    DS8    10
```

# Assembler control directives

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

Directive	Description	Expression restrictions
\$	Includes a file.	
/*comment*/	C-style comment delimiter.	
//	C++style comment delimiter.	
CASEOFF	Disables case sensitivity.	
CASEON	Enables case sensitivity.	
INCLUDE	Includes a file.	
LTORG	Directs the current literal pool to be assembled immediately after the directive.	
RADIX	Sets the default base on all numeric values.	No forward references No external references Absolute Fixed

Table 27: Assembler control directives

## SYNTAX

```
$filename  
/* comment */  
// comment  
CASEOFF  
CASEON  
INCLUDE filename  
LTORG  
RADIX expr
```

## PARAMETERS

<i>comment</i>	Comment ignored by the assembler.
<i>expr</i>	Default base; default 10 (decimal).
<i>filename</i>	Name of file to be included. The \$ character must be the first character on the line.

## DESCRIPTIONS

- Use \$ to insert the contents of a file into the source file at a specified point.
- 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.
- Use LTORG to direct the current literal pool to be assembled. By default, this is done at every END and RSEG directive.

### Controlling case sensitivity

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

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

## EXAMPLES

### Including a source file

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

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

The macro definitions can be included with a \$ directive, as in:

```
NAME include
SECTION MYCODE : CODE (2)
CODE32
; standard macro definitions
$macros.s

; program
main:
        xch R0,R1,R2
```

### Defining comments

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

```
/*
Program to read serial input.
Version 5: 19.2.06
Author: mjp
*/
```

See also, *Comments in C-style preprocessor directives*, page 75.

### Changing the base

To set the default base to 16:

```
SECTION MYCODE : CODE (2)
CODE32
RADIX 16D
MOV R0,#12
```

The immediate argument will then be interpreted as 0x12.

## Controlling case sensitivity

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

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

The following will generate a duplicate label error:

```
CASEOFF

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

## 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.

Table 28: Call frame information directives

Directive	Description
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 28: Call frame information directives (Continued)

## SYNTAX

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

### Names block directives

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

### Extended names block directives

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

### Common block directives

```
CFI COMMON name USING namesblock
CFI ENDCOMMON name
CFI CODEALIGN codealignfactor
CFI DATAALIGN dataalignfactor
```

```

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

```

## Extended common block directives

```

CFI COMMON name EXTENDS commonblock USING namesblock
CFI ENDCOMMON name

```

## Data block directives

```

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

```

## PARAMETERS

<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 91).
<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>section</i>	The name of a section.
<i>size</i>	The size of the frame cell in bytes.
<i>type</i>	The memory type, such as CODE, CONST or DATA. In addition, any of the memory types supported by the IAR ILINK Linker. It is used solely for the purpose of denoting an address space.

## DESCRIPTIONS

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

The parameters are the name of the CFA and the name of the section 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 section 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 section 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 83. For more information on these directives, see *Simple rules*, page 89, and *CFI expressions*, page 91.

## Extending a common block

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

Extend an existing common block with the directive:

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

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

```
CFI ENDCOMMON name
```

## Defining a data block

The *data block* contains the actual tracking information for one continuous piece of code. No section control directive 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 84. For more information on these directives, see *Simple rules*, page 89, and *CFI expressions*, page 91.

## 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 91). 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 83.

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 29: Unary operators in CFI expressions

## Binary operators

Overall syntax: *OPERATOR(operand1, operand2)*

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

Table 30: Binary operators in CFI expressions

Operator	Operands	Description
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 30: 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> .
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 section type <i>type</i> of size <i>size</i> .

Table 31: Ternary operators in CFI expressions

## EXAMPLE

The following is a generic example and not an example specific to the ARM core. 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 32: 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.



# Assembler pseudo-instructions

The ARM IAR Assembler accepts a number of pseudo-instructions, which are translated into correct code. This chapter lists the pseudo-instructions and gives examples of their use.

---

## Summary

In the following table, as well as in the following descriptions:

- ARM denotes pseudo-instructions available after the ARM directive
- CODE16 denotes pseudo-instructions available after the CODE16 directive
- THUMB denotes pseudo-instructions available after the THUMB directive.

**Note:** The properties of THUMB pseudo-instructions depend on whether the used core has the Thumb-2 instruction set or not.

The following table shows a summary of the available pseudo-instructions:

Pseudo-instruction	Directive	Translated to	Description
ADR	ARM	ADD, SUB	Loads a program-relative address into a register.
ADR	CODE16	ADD	Loads a program-relative address into a register.
ADR	THUMB	ADD, SUB	Loads a program-relative address into a register.
ADRL	ARM	ADD, SUB	Loads a program-relative address into a register.
ADRL	THUMB	ADD, SUB	Loads a program-relative address into a register.
LDR	ARM	MOV, MVN, LDR	Loads a register with any 32-bit expression.
LDR	CODE16	MOV, LDR	Loads a register with any 32-bit expression.
LDR	THUMB	MOV, MVN, LDR	Loads a register with any 32-bit expression.

Table 33: Pseudo-instructions

Pseudo-instruction	Directive	Translated to	Description
MOV	CODE16	ADD	Moves the value of a low register to another low register (R0–R7).
MOV32	THUMB	MOV, MOVT	Loads a register with any 32-bit value.
NOP	ARM	MOV	Generates the preferred ARM no-operation code.
NOP	CODE16	MOV	Generates the preferred Thumb no-operation code.

Table 33: Pseudo-instructions (Continued)

## Descriptions of pseudo-instructions

The following section gives reference information about each pseudo-instruction.

---

ADR (ARM) ADR{condition} register,expression

### Parameters

{condition} Can be one of the following: EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE, and AL.

register The register to load.

expression A program location counter-relative expression that evaluates to an address that is not word-aligned within the range -247 to +263 bytes, or a word-aligned address within the range -1012 to +1028 bytes. Unresolved expressions (for example expressions that contain external labels, or labels in other sections) must be within the range -247 to +263 bytes.

### Description

ADR always assembles to one instruction. The assembler attempts to produce a single ADD or SUB instruction to load the address:

```
SECTION MYCODE : CODE (2)
CODE32
ADR r0,thumb ; => ADD r0,pc,#1
BX r0
CODE16
thumb
```

---

ADR (CODE16) ADR *register, expression*

### Parameters

- |                   |  |
|-------------------|--|
| <i>register</i>   | The register to load.  |
| <i>expression</i> | A program-relative expression that evaluates to a word-aligned address within the range +4 to +1024 bytes. |

### Description

In Thumb mode, ADR can generate word-aligned addresses only (that is, addresses divisible by 4). Use the ALIGNROM directive to ensure that the address is aligned (unless DC32 is used, because it is always word aligned):

```
SECTION MYCODE : CODE (2)
ADR r0,my_data ; => ADD r0,pc,#4
ADD r0,r0,r1
BX lr
DATA
ALIGNROM 2
my_data DC32 0xABCD19
END
```

---

ADR (THUMB) ADR{*condition*} *register,expression*

### Parameters

- |                      |  |
|----------------------|--|
| { <i>condition</i> } | An optional condition code if the instruction is placed after an IT instruction.                 |
| <i>register</i>      | The register to load.  |
| <i>expression</i>    | A program-relative expression that evaluates to an address within the range -4095 to 4095 bytes. |

### Description

Similar to ADR (CODE16), but the address range can be larger if a 32-bit Thumb-2 instruction is available in the architecture used. If only 16-bit Thumb instructions are available, see ADR (CODE16), page 99.

If the address offset is positive and the address is word-aligned, the 16-bit ADR (CODE16) version will be generated by default.

The 16-bit version can be specified explicitly with the `ADR.N` instruction. The 32-bit version can be specified explicitly with the `ADR.W` instruction.

`ADRL (ARM) ADRL{condition} register,expression`

### Parameters

<code>{condition}</code>	Can be one of the following: EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE, and AL.
<code>register</code>	The register to load.
<code>expression</code>	A register-relative expression that evaluates to an address that is not word-aligned within 64 Kbytes, or a word-aligned address within 256 Kbytes. Unresolved expressions (for example expressions that contain external labels, or labels in other sections) must be within 64 Kbytes. The address can be either before or after the address of the instruction.

### Description

The `ADRL` pseudo-instruction loads a program-relative address into a register. It is similar to the `ADR` pseudo-instruction. `ADRL` can load a wider range of addresses than `ADR` because it generates two data processing instructions. `ADRL` always assembles to two instructions. Even if the address can be reached in a single instruction, a second, redundant instruction is produced. If the assembler cannot construct the address in two instructions, it generates an error message and the assembly fails.

**Note:** `ADRL` is not available when assembling Thumb instructions. Use it only in ARM code.

### Example

```
SECTION MYCODE : CODE (2)
ADRL r1,my_data+0x2345 ; => ADD r1,pc,#0x45
                           ; => ADD r1,r1,#0x2300
DATA
my_data: DC32 0
END
```

---

ADRL (THUMB) ADRL{condition} register,expression

### Parameters

{condition}	An optional condition code if the instruction is placed after an IT instruction.
register	The register to load.
expression	A program-relative expression that evaluates to an address within the range $\pm 1$ Mbyte.

### Description

Similar to ADRL (ARM), but the address range can be larger. This instruction is only available in a core supporting the Thumb-2 instruction set.

---

LDR (ARM) LDR{condition} register,=expression1

or

LDR{condition} register,expression2

### Parameters

condition	An optional condition code.
register	The register to be loaded.
expression1	Any 32-bit expression.
expression2	A program location counter-relative expression in the range -4087 to +4103 from the program location counter.

### Description

The first form of the LDR pseudo-instruction loads a register with any 32-bit expression. The second form of the instruction reads a 32-bit value from an address specified by the expression. Note that there is also a true LDR instruction.

If the value of expression1 is within the range of a MOV or MVN instruction, the assembler generates the appropriate instruction. If the value of expression1 is not within the range of a MOV or MVN instruction, or if the expression1 is unsolved, the assembler places the constant in a literal pool and generates a program-relative LDR instruction that reads the constant from the literal pool. The offset from the program location counter to the constant must be less than 4 Kbytes. See also the LTORG directive in the section *Assembler control directives*, page 79, for more information.

**Example**

```

SECTION MYCODE : CODE (2)
LDR r1,=0x12345678 ; => LDR r1,[pc,#4]
; loads 0x12345678 from the literal pool into r1
LDR r2,my_data ; loads 0xFFEEDDCC into r2
; => LDR r2,[pc,#-4]
DATA
my_data DC32 0xFFEEDDCC
LTORG
END

```

---

LDR (CODE16) LDR *register*, =*expression1*

or

LDR *register*, *expression2*

**Parameters**

*register* The register to be loaded. LDR can access the low registers (R0–R7) only.

*expression1* Any 32-bit expression.

*expression2* A program location counter-relative expression +4 to +1024 from the program location counter.

**Description**

As in ARM mode, the first form of the LDR pseudo-instruction in Thumb mode loads a register with any 32-bit expression. The second form of the instruction reads a 32-bit value from an address specified by the expression. However, the offset from the program location counter to the constant must be positive and less than 1 Kbyte.

**Example**

```

EXTERN ext_label
SECTION MYCODE : CODE (2)
LDR r1,=ext_label ; => LDR r1,[pc,#8]
; loads ext_label from the literal pool into r1
NOP
LDR r2,my_data ; loads 0xFFEEDDCC into r2
NOP ; => LDR r2,[pc,#0]
DATA
my_data DC32 0xFFEEDDCC
LTORG

```

END

---

LDR (THUMB) LDR{condition} register,=expression

### Parameters

<i>condition</i>	An optional condition code if the instruction is placed after an IT instruction.
<i>register</i>	The register to be loaded.
<i>expression</i>	Any 32-bit expression.

### Description

Similar to the LDR (CODE16) instruction, but by using a 32-bit instruction, a larger value can be loaded directly with a MOV or MVN instruction without requiring the constant to be placed in a literal pool.

If only 16-bit Thumb instructions are available, see *LDR (CODE16)*, page 102.

By specifying a 16-bit version explicitly with the LDR.N instruction, a 16-bit instruction is always generated. This may lead to the constant being placed in the literal pool, even though a 32-bit instruction could have loaded the value directly using MOV or MVN.

By specifying a 32-bit version explicitly with the LDR.W instruction, a 32-bit instruction is always generated.

If you do not specify either .N or .W, the 16-bit LDR (CODE16) instruction will be generated, unless Rd is R8-R15, which leads to the 32-bit variant being generated.

**Note:** The syntax LDR{condition} register, expression2, as described for LDR (ARM) and LDR (CODE16), is no longer considered a pseudo-instruction. It is part of the normal instruction set as specified in the Unified Assembler syntax from Advanced RISC Machines Ltd.

---

MOV (CODE16) MOV Rd, Rs

### Parameters

- Rd              The destination register.  
 Rs              The source register.

### Description

The Thumb `MOV` pseudo-instruction moves the value of a low register to another low register (R0–R7). The Thumb `MOV` instruction cannot move values from one low register to another.

**Note:** The `ADD` immediate instruction generated by the assembler has the side-effect of updating the condition codes.

The `MOV` pseudo-instruction uses an `ADD` immediate instruction with a zero immediate value.

**Note:** This description is only valid when using the `CODE16` directive. After the `THUMB` directive, the interpretation of the instruction syntax is defined by the Undefined Assembler syntax from Advanced RISC Machines Ltd.

### Example

```
MOV r2,r3 ; generates the opcode for ADD r2,r3,#0
```

---

MOV32 (THUMB) MOV32{condition} register,expression

### Parameters

- condition*    An optional condition code if the instruction is placed after an `IT` instruction.  
*register*       The register to be loaded.  
*expression*      Any 32-bit expression.

### Description

Similar to the `LDR (THUMB)` instruction, but will load the constant by generating a pair of the `MOV (MOVW)` and the `MOVT` instructions.

This pseudo-instruction always generates two 32-bit instructions and it is only available in a core supporting the Thumb-2 instruction set.

---

NOP (ARM) NOP

### Description

NOP generates the preferred ARM no-operation code:

```
MOV r0,r0
```

---

NOP (CODE16) NOP

### Description

NOP generates the preferred Thumb no-operation code:

```
MOV r8,r8
```



# Assembler 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 displayed on the screen, as well as printed in the optional list file.

All messages are issued as complete, self-explanatory messages. The message consists of the incorrect source line, with a pointer to where the problem was detected, followed by the source line number and the diagnostic message. If include files are used, error messages will be preceded by the source line number and the name of the *current* file:

```
ADS      B,C  
-----^  
"subfile.h",4  Error[40]: bad instruction
```

---

## Severity levels

The diagnostic messages produced by the ARM IAR Assembler reflect problems or errors that are found in the source code or occur at assembly time.

### OPTIONS FOR DIAGNOSTICS

There are two assembler options for diagnostics. You can:

- Disable or enable all warnings, ranges of warnings, or individual warnings, see *-w*, page 26
- Set the number of maximum errors before the compilation stops, see *-E*, page 19.

### ASSEMBLY WARNING MESSAGES

Assembly warning messages are produced when the assembler has found a construct which is probably the result of a programming error or omission.

### COMMAND LINE ERROR MESSAGES

Command line errors occur when the assembler is invoked with incorrect parameters. The most common situation is when a file cannot be opened, or with duplicate, misspelled, or missing command line options.

## ASSEMBLY ERROR MESSAGES

Assembly error messages are produced when the assembler has found a construct which violates the language rules.

## ASSEMBLY FATAL ERROR MESSAGES

Assembly fatal error messages are produced when the assembler has found a user error so severe that further processing is not considered meaningful. After the diagnostic message has been issued the assembly is immediately terminated. These error messages are identified as `Fatal` in the error messages list.

## ASSEMBLER INTERNAL ERROR MESSAGES

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.

During assembly a number of internal consistency checks are performed and if any of these checks fail, the assembler will terminate after giving a short description of the problem. Such errors should normally not occur. However, if you should encounter an error of this type, it should be reported to your software distributor or to 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.

# Migrating to the ARM IAR Assembler

Assembly source code that was originally written for other assemblers can also be used with the ARM IAR Assembler. The assembler option **-j** allows you to use a number of alternative register names, mnemonics and operators.

This chapter contains information that is useful when migrating from an existing product to the ARM IAR Assembler.

---

## Introduction

The ARM IAR Assembler (IASMARM) was designed using the same look and feel as other IAR assemblers, while still making it easy to translate source code written for the ARMASM assembler from Advanced RISC Machines Ltd.

When the option **-j (Allow alternative register names, mnemonics and operands)** is selected, the instruction syntax is the same in IASMARM as in ARMASM. Many features, such as directives and macros, are, however, incompatible and cause syntax errors. There are also differences in Thumb code labels that may cause problems without generating errors or warnings. Be extra careful when you use such labels in situations other than jumps.

**Note:** For new code, use the ARM IAR Assembler register names, mnemonics and operators.

### THUMB CODE LABELS

Labels placed in Thumb code, i.e. that appear after a **CODE16** directive, always have bit 0 set (i.e. an odd label) in IASMARM. ARMASM, on the other hand, does not set bit 0 on symbols in expressions that are solved at assembly time. In the following example, the symbol **T** is local and placed in Thumb code. It will have bit 0 set when assembled with IASMARM, but not when assembled with ARMASM (except in **DCD**, since it is solved at link time for relocatable sections). Thus, the instructions will be assembled differently.

**Example**

```

SECTION MYCODE : CODE (2)
CODE32
ADR R0,T+1
MOV R1,#T-.
DD      DATA
DCD T
CODE16
T      NOP

```

Rewrite instructions like this to make them portable (i.e. have the same effect when assembled using both IASMARM and ARMASM). Note that ADR is equivalent to an ADD with PC.

```

SECTION MYCODE : CODE (2)
CODE32
ADD R0,PC,#(T-.-8) :OR: 1
MOV R1,#(T-.) :AND: 0xFFFFFFF
DD      DATA
DCD T
CODE16
T      NOP

```

**Alternative register names**

The ARM IAR Assembler will translate the register names below used in other assemblers when the option -j is selected. These alternative register names are allowed in both ARM and Thumb modes. The following table lists the alternative register names and the ARM IAR Assembler register names:

Alternative register name	ARM IAR Assembler register name
A1	R0
A2	R1
A3	R2
A4	R3
V1	R4
V2	R5
V3	R6
V4	R7
V5	R8

Table 34: Alternative register names

Alternative register name	ARM IAR Assembler register name
V6	R9
V7	R10
SB	R9
SL	R10
FP	R11
IP	R12

*Table 34: Alternative register names (Continued)*

For further descriptions of the registers, see *Register symbols*, page 9.

---

## Alternative mnemonics

A number of mnemonics used by other assemblers will be translated by the ARM IAR Assembler when the option `-j` is specified. These alternative mnemonics are allowed in CODE16 mode only. The following table lists the alternative mnemonics:

Alternative mnemonic	ARM IAR Assembler mnemonic
ADCS	ADC
ADDS	ADD
ANDS	AND
ASLS	LSL
ASRS	ASR
BICS	BIC
BNCC	BCS
BNCS	BCC
BNEQ	BNE
BNGE	BLT
BNGT	BLE
BNHI	BLS
BNLE	BGT
BNLO	BCS
BNLS	BHI
BNLT	BGE
BNMI	BPL

*Table 35: Alternative mnemonics*

Alternative mnemonic	ARM IAR Assembler mnemonic
BNNE	BEQ
BNPL	BMI
BNVC	BVS
BNVS	BVC
CMN{cond}S	CMN{cond}
CMP{cond}S	CMP{cond}
EORS	EOR
LSLS	LSL
LSRS	LSR
MOVS	MOV
MULS	MUL
MVNS	MVN
NEGS	NEG
ORRS	ORR
RORS	ROR
SBCS	SBC
SUBS	SUB
TEQ{cond}S	TEQ{cond}
TST{cond}S	TST{cond}

Table 35: Alternative mnemonics (Continued)

Refer to the *ARM Architecture Reference Manual* (Prentice-Hall) for full descriptions of the mnemonics.

---

## Operator synonyms

A number of operators used by other assemblers will be translated by the ARM IAR Assembler when the option `-j` is specified. The following operator synonyms are allowed in both ARM and Thumb modes:

Operator synonym	ARM IAR Assembler operator
:AND:	&
:EOR:	^
:LAND:	&&
:LEOR:	XOR

Table 36: Operator synonyms

Operator synonym	ARM IAR Assembler operator
:LNOT:	!
:LOR:	
:MOD:	%
:NOT:	~
:OR:	
:SHL:	<<
:SHR:	>>

Table 36: Operator synonyms (Continued)

**Note:** ARM IAR Assembler operators and operator synonyms have different precedence levels. For further descriptions of the operators, see the chapter *Assembler operators*, page 29.

## Warning messages

Unless the option `-j` is specified, the ARM IAR Assembler will issue warning messages when the alternative names are used, or when illegal combinations of operands are encountered. The following sections list the warning messages:

### The first register operand omitted

The first register operand was missing in an instruction that requires three operands, where the first two are unindexed registers (ADD, SUB, LSL, LSR, and ASR).

### The first register operand duplicated

The first register operand was a register that was included in the operation, and was also a destination register.

Example of incorrect code:

```
MUL R0, R0, R1
```

Example of correct code:

```
MUL R0, R1
```

### Immediate #0 omitted in Load/Store

Immediate #0 was missing in a load/store instruction.

Example of incorrect code:

```
LDR R0, [R1]
```

Example of correct code:

```
LDR R0, [R1,#0]
```

# A

AAPCS (assembler directive) . . . . .	47
absolute expressions . . . . .	11
ADD (assembler instruction) . . . . .	98
ADD (CFI operator) . . . . .	92
addition (assembler operator) . . . . .	32
address field, in assembler list file . . . . .	13
addresses, loading into a register. . . . .	98–101
ADR (ARM) (pseudo-instruction) . . . . .	98
ADR (CODE16) (pseudo-instruction) . . . . .	99
ADR (THUMB) (pseudo-instruction) . . . . .	99
ADRL (ARM) (pseudo-instruction) . . . . .	100
ADRL (THUMB) (pseudo-instruction) . . . . .	101
ALIAS (assembler directive) . . . . .	57
alignment, of sections . . . . .	56
ALIGNRAM (assembler directive) . . . . .	54
ALIGNROM (assembler directive) . . . . .	54
:AND: (assembler operator) . . . . .	35
AND (CFI operator) . . . . .	92
architecture, ARM . . . . .	xi
_args (predefined macro symbol) . . . . .	63
ARM architecture and instruction set . . . . .	xi
ARM (assembler directive) . . . . .	52
ARMASM assembler . . . . .	109
_ARMVFP_ (predefined symbol) . . . . .	10
ASCII character constants. . . . .	7
asm (filename extension) . . . . .	3
assembler	
invocation syntax . . . . .	3
assembler control directives . . . . .	79
assembler diagnostics . . . . .	107
assembler directives	
assembler control . . . . .	79
call frame information . . . . .	82
CFI directives . . . . .	82
conditional assembly . . . . .	58
<i>See also</i> C-style preprocessor directives	
C-style preprocessor . . . . .	72

data definition or allocation . . . . .	77
list file control . . . . .	68
macro processing . . . . .	60
module control . . . . .	47
section control . . . . .	54
summary . . . . .	43
symbol control . . . . .	50
value assignment . . . . .	57
assembler environment variables . . . . .	4
assembler expressions . . . . .	6
assembler EXTRN (assembler directive) . . . . .	50
assembler instructions . . . . .	6
ADD . . . . .	98
BX . . . . .	53
LDR . . . . .	101
MOV . . . . .	101
MVN . . . . .	101
SUB . . . . .	98
assembler labels . . . . .	9
format of . . . . .	5
in Thumb code . . . . .	109
assembler list files	
address field . . . . .	13
comments . . . . .	80
conditional code and strings . . . . .	69
conditions, specifying . . . . .	17
cross-references, generating . . . . .	26, 70
data field . . . . .	13
disabling . . . . .	69
enabling . . . . .	69
filename, specifying . . . . .	22
format, specifying . . . . .	70
generated lines, controlling . . . . .	69
generating . . . . .	21
header section, omitting . . . . .	23
#include files, specifying . . . . .	21
lines per page, specifying . . . . .	24
macro execution information, including . . . . .	17
macro-generated lines, controlling . . . . .	69

symbol and cross-reference table	13
tab spacing, specifying	25
using directives to format	70
assembler macros	
arguments, passing to	63
defining	62
generated lines, controlling in list file	69
in-line routines	65
predefined symbol	63
processing	64
quote characters, specifying	22
special characters, using	63
assembler object file, specifying filename	23
assembler operators	29
in expressions	6
precedence	29
summary	29
synonyms	31
assembler options	
passing to assembler	4
command line, setting	15
extended command file, setting	15
summary	16
typographic convention	xiii
assembler output, including debug information	24
assembler pseudo-instructions	97
assembler source code, porting	53
assembler source files, including	74, 81
assembler source format	5
assembler symbols	8
exporting	51
importing	51
in relocatable expressions	11
predefined	10
undefining	25
assembly error messages	108
assembly messages format	107
assembly warning messages	107
disabling	26

ASSIGN (assembler directive)	57
assumptions (programming experience)	xi

## B

-B (assembler option)	17
backtrace information, defining	82
bitwise AND (assembler operator)	35
bitwise exclusive OR (assembler operator)	36
bitwise NOT (assembler operator)	35
bitwise OR (assembler operator)	36
_BUILD_NUMBER_ (predefined symbol)	10
BX (assembler instruction)	53
byte order	10
specifying	19
BYTE1 (assembler operator)	37
BYTE2 (assembler operator)	37
BYTE3 (assembler operator)	37
BYTE4 (assembler operator)	37

## C

-c (assembler option)	17
call frame information directives	82
case sensitive user symbols	25
case sensitivity, controlling	80
CASEOFF (assembler directive)	79
CASEON (assembler directive)	79
CFI directives	82
CFI expressions	91
CFI operators	92
character constants, ASCII	7
CODE16 (assembler directive)	52
CODE32 (assembler directive)	52
COL (assembler directive)	68
command line error messages, assembler	107
command line options	15
part of invocation syntax	3
passing	4

command line, extending	19
comments	
in assembler list file	80
in assembler source code	5
multi-line, using with assembler directives	81
comments, in C-style preprocessor directives	75
COMPLEMENT (CFI operator)	92
computer style, typographic convention	xiii
conditional assembly directives	58
<i>See also</i> C-style preprocessor directives	
conditional code and strings, listing	69
conditional list file	17
constants, default base of	80
constants, integer	6
conventions, typographic	xiii
--cpu (assembler option)	18
CPU, defining in assembler. <i>See</i> processor configuration	
CRC, in assembler list file	13
cross-references, in assembler list file	70
generating	26
current time/date (assembler operator)	38
C-style preprocessor directives	72

## D

-D (assembler option)	18
data allocation directives	77
data definition directives	77
data field, in assembler list file	13
DATA (assembler directive)	52
data, defining in Thumb code section	53
_DATE_ (predefined symbol)	10
DATE (assembler operator)	38
DCB (assembler directive)	77
DCD (assembler directive)	77
DCW (assembler directive)	77
DC8 (assembler directive)	77
DC16 (assembler directive)	77
DC24 (assembler directive)	77

DC32 (assembler directive)	77
debug information, including in assembler output	24
default base, for constants	80
#define (assembler directive)	72
DEFINE (assembler directive)	57
DF32 (assembler directive)	77
DF64 (assembler directive)	77
diagnostics	107
options for	107
directives. <i>See</i> assembler directives	
DIV (CFI operator)	92
division (assembler operator)	33
document conventions	xiii
DS8 (assembler directive)	77
DS16 (assembler directive)	77
DS24 (assembler directive)	77
DS32 (assembler directive)	77

## E

-E (assembler option)	19
-e (assembler option)	19
efficient coding techniques	14
#elif (assembler directive)	72
#else (assembler directive)	72
ELSE (assembler directive)	58
ELSEIF (assembler directive)	58
END (assembler directive)	47
--endianness (assembler option)	19
endianess. <i>See</i> byte order	
#endif (assembler directive)	72
ENDIF (assembler directive)	58
ENDM (assembler directive)	60
ENDR (assembler directive)	60
environment variables	
assembler	4
IASMARM	4
IASMARM_INC	4
:EOR: (assembler operator)	36

EQ (CFI operator).....	92
EQU (assembler directive) .....	57
equal (assembler operator) .....	34
#error (assembler directive) .....	72
error messages	
format .....	107
maximum number, specifying .....	19
#error, using to display.....	75
EVEN (assembler directive) .....	54
EXITM (assembler directive) .....	60
experience, programming .....	xi
expressions .....	6
extended command line file .....	15, 19
EXTERN (assembler directive).....	50
EXTRN (assembler directive).....	50
EXTWEAK (assembler directive).....	50

## F

-f (assembler option).....	15, 19
false value, in assembler expressions .....	8
fatal errors.....	108
_FILE_ (predefined symbol).....	10
file extensions. <i>See</i> filename extensions	
file types	
assembler source .....	3
extended command line .....	15, 19
#include .....	20
filename extensions	
asm .....	3
msa .....	3
s .....	3
xcl .....	15, 19
filenames, specifying for assembler object file .....	23–24
first byte (assembler operator).....	37
floating-point constants.....	7
format, assembler source code .....	5
fourth byte (assembler operator).....	37
--fpu (assembler option) .....	20

fractions .....	8
FRAME (CFI operator).....	93

## G

-G (assembler option) .....	20
GE (CFI operator).....	92
global value, defining .....	58
glossary.....	xi
greater than or equal (assembler operator) .....	35
greater than (assembler operator) .....	34
GT (CFI operator).....	92

## H

header files, SFR.....	14
header section, omitting from assembler list file.....	23
high byte (assembler operator) .....	38
high word (assembler operator) .....	38
HIGH (assembler operator).....	38
HWRD (assembler operator) .....	38

## I

-I (assembler option).....	20
-i (assembler option).....	21
_IAR_SYSTEMS_ASM_ (predefined symbol) .....	10
_IASMARM_ (predefined symbol) .....	10
IASMARM (environment variable) .....	4
IASMARM_INC (environment variable) .....	4
#if (assembler directive) .....	72
IF (assembler directive) .....	59
IF (CFI operator).....	93
#ifdef (assembler directive).....	73
#ifndef (assembler directive).....	73
IMPORT (assembler directive) .....	50
#include files.....	20–21
#include (assembler directive) .....	73
include paths, specifying .....	20

INCLUDE (assembler directive) . . . . .	79
instruction set, ARM . . . . .	xi
integer constants . . . . .	6
internal errors, assembler . . . . .	108
invocation syntax . . . . .	3
in-line coding, using macros . . . . .	65
iomacros.h . . . . .	14

## J

-j (assembler option) . . . . .	21
---------------------------------	----

## L

-L (assembler option) . . . . .	21
-l (assembler option) . . . . .	22
labels. <i>See</i> assembler labels	
:LAND: (assembler operator) . . . . .	35
LDR (ARM) (pseudo-instruction) . . . . .	101
LDR (assembler instruction) . . . . .	101
LDR (CODE16) (pseudo-instruction) . . . . .	102
LDR (THUMB) (pseudo-instruction) . . . . .	103
LE (CFI operator) . . . . .	92
:LEOR: (assembler operator) . . . . .	41
less than or equal (assembler operator) . . . . .	34
less than (assembler operator) . . . . .	33
LIBRARY (assembler directive) . . . . .	46
_LINE_ (predefined symbol) . . . . .	10
#line (assembler directive) . . . . .	73
lines per page, in assembler list file . . . . .	24
list file format . . . . .	13
body . . . . .	13
CRC . . . . .	13
header . . . . .	13
symbol and cross reference	
listing control directives . . . . .	68
literal pool . . . . .	101
LITERAL (CFI operator) . . . . .	92
_LITTLE_ENDIAN_ (predefined symbol) . . . . .	10

:LNOT: (assembler operator) . . . . .	36
LOAD (CFI operator) . . . . .	93
local value, defining . . . . .	58
LOCAL (assembler directive) . . . . .	60
logical AND (assembler operator) . . . . .	35
logical exclusive OR (assembler operator) . . . . .	41
logical NOT (assembler operator) . . . . .	36
logical shift left (assembler operator) . . . . .	40
logical shift right (assembler operator) . . . . .	41
:LOR: (assembler operator) . . . . .	37
low byte (assembler operator) . . . . .	39
low register values, moving . . . . .	104
low word (assembler operator) . . . . .	39
LOW (assembler operator) . . . . .	39
LSHIFT (CFI operator) . . . . .	92
LSTCND (assembler directive) . . . . .	68
LSTCOD (assembler directive) . . . . .	68
LSTEXP (assembler directives) . . . . .	68
LSTMAC (assembler directive) . . . . .	68
LSTOUT (assembler directive) . . . . .	68
LSTPAG (assembler directive) . . . . .	68
LSTREP (assembler directive) . . . . .	68
LSTXRF (assembler directive) . . . . .	68
LT (CFI operator) . . . . .	92
LTORG (assembler directive) . . . . .	79
LWRD (assembler operator) . . . . .	39

## M

-M (assembler option) . . . . .	22
macro execution information, including in list file . . . . .	17
macro processing directives . . . . .	60
macro quote characters . . . . .	63
specifying . . . . .	22
MACRO (assembler directive) . . . . .	60
macros. <i>See</i> assembler macros	
memory space, reserving and initializing . . . . .	78
memory, reserving space in . . . . .	77
#message (assembler directive) . . . . .	73

messages, excluding from standard output stream	24
migration to the ARM IAR Assembler	109
alternative mnemonics	111
alternative register names	110
operator synonyms	112
warning messages	113
:MOD: (assembler operator)	36
MOD (CFI operator)	92
mode control directives	52
module consistency	49
module control directives	47
modules, beginning	48
modulo (assembler operator)	36
MOV (assembler instruction)	101
MOV (CODE16) (pseudo-instruction)	104
MOV (THUMB) (pseudo-instruction)	104
msa (filename extension)	3
MUL (CFI operator)	92
multibyte character support	23
multiplication (assembler operator)	32
MVN (assembler instruction)	101

## N

-N (assembler option)	23
-n (assembler option)	23
NAME (assembler directive)	47
NE (CFI operator)	92
NOP (ARM) (pseudo-instruction)	105
NOP (CODE16) (pseudo-instruction)	105
:NOT: (assembler operator)	35
not equal (assembler operator)	34
NOT (CFI operator)	92
no-operation code, generating	105

## O

-O (assembler option)	23
-o (assembler option)	24

ODD (assembler directive)	54
operands	
format of	5
in assembler expressions	6
operations, format of	5
operation, silent	24
operators. <i>See</i> assembler operators	
option summary	16
:OR: (assembler operator)	36
OR (CFI operator)	92
OVERLAY (assembler directive)	50

## P

-p (assembler option)	24
PAGE (assembler directive)	68
PAGSIZ (assembler directive)	68
parameters, typographic convention	xiii
porting assembler source code	53
#pragma (assembler directive)	73
precedence, of assembler operators	29
predefined register symbols	9
predefined symbols	10
in assembler macros	63
undefining	25
preprocessor symbols	
defining	18
defining and undefining	74
prerequisites (programming experience)	xi
PRESERVE8 (assembler directive)	47
processor mode, directives	52
program location counter (PLC)	9
PROGRAM (assembler directive)	47
programming experience, required	xi
programming hints	14
pseudo-instructions	97
PUBLIC (assembler directive)	50
PUBWEAK (assembler directive)	50

**R**

-r (assembler option) . . . . .	24
RADIX (assembler directive) . . . . .	79
reference information, typographic convention . . . . .	xiii
registered trademarks . . . . .	ii
registers . . . . .	9
alternative names of . . . . .	110
relocatable expressions . . . . .	11
relocatable sections, beginning . . . . .	55
repeating statements . . . . .	64
REPT (assembler directive) . . . . .	61
REPTC (assembler directive) . . . . .	61
REPTI (assembler directive) . . . . .	61
REQUIRE (assembler directive) . . . . .	50
REQUIRE8 (assembler directive) . . . . .	48
RSEG (assembler directive) . . . . .	54
RSHIFTA (CFI operator) . . . . .	92
RSHIFTL (CFI operator) . . . . .	93
RTMODEL (assembler directive) . . . . .	48
rules, in CFI directives . . . . .	89
runtime model attributes, declaring . . . . .	49

**S**

-S (assembler option) . . . . .	24
-s (assembler option) . . . . .	25
s (filename extension) . . . . .	3
second byte (assembler operator) . . . . .	37
section begin (assembler operator) . . . . .	39
section control directives . . . . .	54
section end (assembler operator) . . . . .	40
SECTION (assembler directive) . . . . .	54
sections	
aligning . . . . .	56
relocatable . . . . .	55
SECTION_TYPE (assembler directive) . . . . .	54
SET (assembler directive) . . . . .	57
SETA (assembler directive) . . . . .	57

SFB (assembler operator) . . . . .	39
SFE (assembler operator) . . . . .	40
SFR. <i>See</i> special function registers	
:SHL: (assembler operator) . . . . .	40
:SHR: (assembler operator) . . . . .	41
silent operation, specifying in assembler . . . . .	24
simple rules, in CFI directives . . . . .	89
source files, including . . . . .	74, 81
source format, assembler . . . . .	5
special function registers . . . . .	14
standard input stream (stdin), reading from . . . . .	20
standard output stream, disabling messages to . . . . .	24
statements, repeating . . . . .	64
SUB (assembler instruction) . . . . .	98
SUB (CFI operator) . . . . .	93
subtraction (assembler operator) . . . . .	33
symbol and cross-reference table, in assembler list file . . . . .	13

*See also* Include cross-reference

symbol control directives . . . . .	50
symbols	
<i>See also</i> assembler symbols	
exporting to other modules . . . . .	51
predefined, in assembler . . . . .	10
predefined, in assembler macro . . . . .	63
user-defined, case sensitive . . . . .	25
synonyms, of assembler operators . . . . .	31
syntax conventions . . . . .	xiii

**T**

-t (assembler option) . . . . .	25
tab spacing, specifying in assembler list file . . . . .	25
target core, specifying. <i>See</i> processor configuration	
temporary values, defining . . . . .	58
terminology . . . . .	xi
third byte (assembler operator) . . . . .	37
THUMB (assembler directive) . . . . .	52
-TID_ _ (predefined symbol) . . . . .	10
--TIME_ _ (predefined symbol) . . . . .	10

time-critical code .....	65
trademarks .....	ii
true value, in assembler expressions .....	8
typographic conventions .....	xiii

## U

-U (assembler option) .....	25
UGT (assembler operator) .....	41
ULT (assembler operator) .....	41
UMINUS (CFI operator) .....	92
unary minus (assembler operator) .....	32
unary plus (assembler operator) .....	32
#undef (assembler directive) .....	73
unsigned greater than (assembler operator) .....	41
unsigned less than (assembler operator) .....	41
user symbols, case sensitive .....	25

## V

value assignment directives .....	57
values, defining .....	77
VAR (assembler directive) .....	57
_VER_ (predefined symbol) .....	10

## W

-w (assembler option) .....	26
warnings .....	107
disabling .....	26

## X

-x (assembler option) .....	26
xcl (filename extension) .....	15, 19
XOR (assembler operator) .....	41
XOR (CFI operator) .....	93

# Symbols

! (assembler operator) .....	36
!= (assembler operator) .....	34
#define (assembler directive) .....	72
#elif (assembler directive) .....	72
#else (assembler directive) .....	72
#endif (assembler directive) .....	72
#error (assembler directive) .....	72
#if (assembler directive) .....	72
#ifdef (assembler directive) .....	73
#ifndef (assembler directive) .....	73
#include files .....	20–21
#include (assembler directive) .....	73
#line (assembler directive) .....	73
#message (assembler directive) .....	73
#pragma (assembler directive) .....	73
#undef (assembler directive) .....	73
\$ (assembler directive) .....	79
\$ (program location counter) .....	9
% (assembler operator) .....	36
& (assembler operator) .....	35
&& (assembler operator) .....	35
* (assembler operator) .....	32
+ (assembler operator) .....	32
- (assembler operator) .....	32–33
-B (assembler option) .....	17
-c (assembler option) .....	17
-D (assembler option) .....	18
-E (assembler option) .....	19
-e (assembler option) .....	19
-f (assembler option) .....	15, 19
-G (assembler option) .....	20
-I (assembler option) .....	20
-i (assembler option) .....	21
-j (assembler option) .....	21, 109
-L (assembler option) .....	21
-l (assembler option) .....	22
-M (assembler option) .....	22

-N (assembler option) . . . . .	23
-n (assembler option) . . . . .	23
-O (assembler option) . . . . .	23
-o (assembler option) . . . . .	24
-p (assembler option) . . . . .	24
-r (assembler option) . . . . .	24
-S (assembler option) . . . . .	24
-s (assembler option) . . . . .	25
-t (assembler option) . . . . .	25
-U (assembler option) . . . . .	25
-w (assembler option) . . . . .	26
-x (assembler option) . . . . .	26
--cpu (assembler option) . . . . .	18
--endian (assembler option) . . . . .	19
--fpu (assembler option) . . . . .	20
/ (assembler operator) . . . . .	33
/*...*/ (assembler directive) . . . . .	79
// (assembler directive) . . . . .	79
< (assembler operator) . . . . .	33
<< (assembler operator) . . . . .	40
<= (assembler operator) . . . . .	34
<> (assembler operator) . . . . .	34
= (assembler directive) . . . . .	57
= (assembler operator) . . . . .	34
== (assembler operator) . . . . .	34
> (assembler operator) . . . . .	34
>= (assembler operator) . . . . .	35
>> (assembler operator) . . . . .	41
^ (assembler operator) . . . . .	36
_ARMVFP_ (predefined symbol) . . . . .	10
_BUILD_NUMBER_ (predefined symbol) . . . . .	10
_DATE_ (predefined symbol) . . . . .	10
_FILE_ (predefined symbol) . . . . .	10
_IAR_SYSTEMS_ASM_ (predefined symbol) . . . . .	10
_LINE_ (predefined symbol) . . . . .	10
_LITTLE_ENDIAN_ (predefined symbol) . . . . .	10
_TID_ (predefined symbol) . . . . .	10
_TIME_ (predefined symbol) . . . . .	10
_VER_ (predefined symbol) . . . . .	10
_args (predefined macro symbol) . . . . .	63
_IASMARM_ (predefined symbol) . . . . .	10
(assembler operator) . . . . .	36
(assembler operator)logical OR (assembler operator) . . . . .	37
~ (assembler operator) . . . . .	35
32-bit expressions, loading in register . . . . .	101

## Numerics