

# IAR Embedded Workbench®

IAR Assembler™ Reference Guide

for Advanced RISC Machines Ltd's  
ARM Cores



AARM-9

**IAR**  
SYSTEMS

## **COPYRIGHT NOTICE**

© 1999–2012 IAR Systems AB.

No part of this document may be reproduced without the prior written consent of IAR Systems AB. 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 Systems, IAR Embedded Workbench, C-SPY, visualSTATE, The Code to Success, IAR KickStart Kit, I-jet, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

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

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

## **EDITION NOTICE**

Ninth edition: May 2012

Part number: AARM-9

This guide applies to version 6.x of IAR Embedded Workbench® for ARM.

Internal reference: M12, asrct2010.3, V\_111012, asrcarm6.40, IMAE.

# Contents

Tables .....	9
Preface .....	11
<b>Who should read this guide</b> .....	11
<b>How to use this guide</b> .....	11
<b>What this guide contains</b> .....	12
<b>Other documentation</b> .....	12
User and reference guides .....	12
The online help system .....	13
Web sites .....	13
<b>Document conventions</b> .....	14
Typographic conventions .....	14
Naming conventions .....	15
Introduction to the IAR Assembler for ARM .....	17
<b>Introduction to assembler programming</b> .....	17
Getting started .....	18
<b>Modular programming</b> .....	18
<b>External interface details</b> .....	19
Assembler invocation syntax .....	19
Passing options .....	20
Environment variables .....	20
Error return codes .....	20
<b>Source format</b> .....	21
<b>Assembler instructions</b> .....	21
<b>Expressions, operands, and operators</b> .....	22
Integer constants .....	22
ASCII character constants .....	23
Floating-point constants .....	23
TRUE and FALSE .....	24
Symbols .....	24
Labels .....	24

Register symbols .....	25
Predefined symbols .....	25
Absolute and relocatable expressions .....	28
Expression restrictions .....	28
<b>List file format</b> .....	29
Header .....	29
Body .....	29
Summary .....	30
Symbol and cross-reference table .....	30
<b>Programming hints</b> .....	30
Accessing special function registers .....	30
Using C-style preprocessor directives .....	31
<b>Assembler options</b> .....	33
<b>Using command line assembler options</b> .....	33
Extended command line file .....	33
<b>Summary of assembler options</b> .....	34
<b>Description of assembler options</b> .....	35
<b>Assembler operators</b> .....	49
<b>Precedence of assembler operators</b> .....	49
<b>Summary of assembler operators</b> .....	49
Parenthesis operator – 1 .....	49
Unary operators – 1 .....	49
Multiplicative arithmetic operators – 2 .....	50
Additive arithmetic operators – 3 .....	50
Shift operators – 4 .....	50
AND operators – 5 .....	50
OR operators – 6 .....	51
Comparison operators – 7 .....	51
Operator synonyms .....	51
<b>Description of assembler operators</b> .....	52

Assembler directives .....	63
<b>Summary of assembler directives</b> .....	63
<b>Module control directives</b> .....	67
Syntax .....	68
Parameters .....	68
Descriptions .....	68
<b>Symbol control directives</b> .....	70
Syntax .....	70
Parameters .....	71
Descriptions .....	71
Examples .....	72
<b>Mode control directives</b> .....	72
Syntax .....	73
Description .....	73
Examples .....	73
<b>Section control directives</b> .....	74
Syntax .....	75
Parameters .....	75
Descriptions .....	76
Examples .....	77
<b>Value assignment directives</b> .....	78
Syntax .....	78
Parameters .....	78
Descriptions .....	78
Examples .....	79
<b>Conditional assembly directives</b> .....	80
Syntax .....	81
Parameters .....	81
Descriptions .....	81
Examples .....	81
<b>Macro processing directives</b> .....	82
Syntax .....	82
Parameters .....	83

Descriptions .....	83
Examples .....	87
<b>Listing control directives .....</b>	<b>90</b>
Syntax .....	90
Parameters .....	91
Descriptions .....	91
Examples .....	92
<b>C-style preprocessor directives .....</b>	<b>94</b>
Syntax .....	95
Parameters .....	95
Descriptions .....	96
Examples .....	98
<b>Data definition or allocation directives .....</b>	<b>99</b>
Syntax .....	100
Parameters .....	100
Descriptions .....	101
Examples .....	101
<b>Assembler control directives .....</b>	<b>102</b>
Syntax .....	103
Parameters .....	103
Descriptions .....	103
Examples .....	104
<b>Call frame information directives .....</b>	<b>105</b>
Syntax .....	107
Parameters .....	108
Descriptions .....	109
Rules for simple cases .....	112
Using expressions for complex cases .....	114
Stack usage analysis directives .....	117
Example .....	117

Assembler pseudo-instructions .....	121
<b>Summary</b> .....	121
<b>Descriptions of pseudo-instructions</b> .....	122
Assembler diagnostics .....	131
<b>Message format</b> .....	131
<b>Severity levels</b> .....	131
Options for diagnostics .....	131
Assembly warning messages .....	131
Command line error messages .....	131
Assembly error messages .....	132
Assembly fatal error messages .....	132
Assembler internal error messages .....	132
Migrating to the IAR Assembler for ARM .....	133
<b>Introduction</b> .....	133
Thumb code labels .....	133
<b>Alternative register names</b> .....	134
<b>Alternative mnemonics</b> .....	135
<b>Operator synonyms</b> .....	136
<b>Warning messages</b> .....	137
Index .....	139





# Tables

1: Typographic conventions used in this guide .....	14
2: Naming conventions used in this guide .....	15
3: Assembler environment variables .....	20
4: Assembler error return codes .....	20
5: Integer constant formats .....	22
6: ASCII character constant formats .....	23
7: Floating-point constants .....	23
8: Predefined register symbols .....	25
9: Predefined symbols .....	26
10: Symbol and cross-reference table .....	30
11: Assembler options summary .....	34
12: Operator synonyms .....	51
13: Assembler directives summary .....	63
14: Module control directives .....	67
15: Symbol control directives .....	70
16: Mode control directives .....	72
17: Section control directives .....	74
18: Value assignment directives .....	78
19: Conditional assembly directives .....	80
20: Macro processing directives .....	82
21: Listing control directives .....	90
22: C-style preprocessor directives .....	94
23: Data definition or allocation directives .....	99
24: Assembler control directives .....	102
25: Call frame information directives .....	106
26: Unary operators in CFI expressions .....	115
27: Binary operators in CFI expressions .....	115
28: Ternary operators in CFI expressions .....	116
29: Code sample with backtrace rows and columns .....	117
30: Pseudo-instructions .....	121
31: Alternative register names .....	134

32: Alternative mnemonics .....	135
33: Operator synonyms .....	136

# Preface

Welcome to the IAR Assembler™ Reference Guide for ARM. The purpose of this guide is to provide you with detailed reference information that can help you to use the IAR Assembler for ARM 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 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 IAR Assembler, you should read the chapter *Introduction to the IAR Assembler for ARM* 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 *IDE Project Management and Building Guide for ARM*. They give product overviews, and tutorials that can help you get started.

---

## What this guide contains

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

- *Introduction to the IAR Assembler for ARM* 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 IAR Assembler for ARM* contains information that is useful when you want to use the IAR Assembler for ARM with source code that was originally developed for another assembler.

---

## Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the Information Center or from the **Help** menu in the IAR Embedded Workbench IDE. The online help system is also available via the F1 key.

### USER AND REFERENCE GUIDES

The complete set of IAR Systems development tools is described in a series of guides. For information about:

- System requirements and information about how to install and register the IAR Systems products, refer to the booklet *Quick Reference* (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, see the guide *Getting Started with IAR Embedded Workbench®*.

- Using the IDE for project management and building, see the *IDE Project Management and Building Guide for ARM*.
- Using the IAR C-SPY® Debugger, see the *C-SPY® Debugging Guide for ARM*.
- Programming for the IAR C/C++ Compiler for ARM and linking using the IAR ILINK Linker, see the *IAR C/C++ Development Guide for ARM*.
- Using the IAR DLIB Library, see the *DLIB Library Reference information*, available in the online help system.
- Using the IAR CLIB Library, see the *IAR C Library Functions Reference Guide*, available in the online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for ARM, see the *IAR Embedded Workbench® Migration Guide for ARM*.
- Developing safety-critical applications using the MISRA C guidelines, see the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.

**Note:** Additional documentation might be available depending on your product installation.

## THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Comprehensive information about debugging using the IAR C-SPY® Debugger
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- Keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1. Note that if you select a function name in the editor window and press F1 while using the CLIB library, you will get reference information for the DLIB library.

## WEB SITES

Recommended web sites:

- The Advanced RISC Machines Ltd web site, **www.arm.com**, that contains information and news about the ARM cores.
- The IAR Systems web site, **www.iar.com**, that holds application notes and other product information.

## Document conventions

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

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

### TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:





Style	Used for
<code>computer</code>	<ul style="list-style-type: none"> <li>• Source code examples and file paths.</li> <li>• Text on the command line.</li> <li>• Binary, hexadecimal, and octal numbers.</li> </ul>
<i>parameter</i>	A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file.
[option]	An optional part of a command.
[a b c]	An optional part of a command with alternatives.
{a b c}	A mandatory part of a command with alternatives.
<b>bold</b>	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none"> <li>• A cross-reference within this guide or to another guide.</li> <li>• Emphasis.</li> </ul>
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide

## NAMING CONVENTIONS

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

<b>Brand name</b>	<b>Generic term</b>
IAR Embedded Workbench® for ARM	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for ARM	the IDE
IAR C-SPY® Debugger for ARM	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for ARM	the compiler
IAR Assembler™ for ARM	the assembler
IAR ILINK Linker™	ILINK, the linker
IAR DLIB Library™	the DLIB library
IAR CLIB Library™	the CLIB library

*Table 2: Naming conventions used in this guide*





# Introduction to the IAR Assembler for ARM

This chapter contains these sections:

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

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 might be situations where you 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 Information Center
- Read about the assembler language interface—also useful when mixing C and assembler modules—in the *IAR C/C++ Development Guide for ARM*
- In the IAR Embedded Workbench IDE, you can base a new project on a *template* for an assembler project.

---

## Modular programming

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

- efficient program development
- reuse of modules
- maintenance.

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

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

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

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

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

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

---

## External interface details

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

You can use the assembler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *IDE Project Management and Building Guide for ARM* 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 this command to generate an object file with debug information:

```
iasmarm prog -r
```

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

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

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

## PASSING OPTIONS

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

- Directly from the command line  
Specify the options on the command line after the `iasmarm` command; see *Assembler invocation syntax*, page 19.
- Via environment variables  
The assembler automatically appends the value of the environment variables to every command line, so it provides a convenient method of specifying options that are required for every assembly; see *Environment variables*, page 20.
- Via a text file by using the `-f` option; see *-f*, page 38.

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

## ENVIRONMENT VARIABLES

You can use these environment variables with the IAR Assembler:

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

Table 3: Assembler environment variables

For example, setting this environment variable always generates 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 *IAR C/C++ Development Guide for ARM*.

## ERROR RETURN CODES

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

Return code	Description
0	Assembly successful, warnings might appear.
1	Warnings occurred (only if the <code>-ws</code> option is used).

Table 4: Assembler error return codes

Return code	Description
2	Errors occurred.

Table 4: Assembler error return codes (Continued)

## Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

<i>label</i>	A definition of a label, which is a symbol that represents an address. If the label starts in the first column—that is, at the far left on the line—the : (colon) is optional.
<i>operation</i>	An assembler instruction or directive. This must not start in the first column—there must be some whitespace to the left of it.
<i>operands</i>	An assembler instruction or directive can have zero, one, or more operands. The operands are separated by commas.
<i>comment</i>	Comment, preceded by a ; (semicolon) C or C++ comments are also allowed.

The components are separated by spaces or tabs.

A source line cannot exceed 2047 characters.

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

## Assembler instructions

The IAR Assembler for ARM 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 results in an error on cores with word alignment.

## Expressions, operands, and operators

Expressions consist of expression operands and operators.

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

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators; see also *Assembler operators*, page 49.

These operands are valid in an expression:

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

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

**Note:** You cannot have two symbols in one expression, or any other complex expression, unless the expression can be resolved at assembly time. If they are not resolved, the assembler generates an error.

### INTEGER CONSTANTS

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

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

Commas and decimal points are not permitted.

The following types of number representation are supported:

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

*Table 5: Integer constant formats*

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

## ASCII CHARACTER CONSTANTS

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

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

Table 6: ASCII character constant formats

## FLOATING-POINT CONSTANTS

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

Floating-point numbers can be written in the format:

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

This table shows some valid examples:

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

Table 7: Floating-point constants

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

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

## 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. For more information, see -s, page 46.

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

Note that symbols and labels are byte addresses.

## LABELS

Symbols used for memory locations are referred to as labels.

### Program location counter (PLC)

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



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

```
section MYCODE:CODE(2)
arm
b . ; Loop forever
end
```

## REGISTER SYMBOLS

This table shows the existing predefined register symbols:

Name	Size	Description
CPSR	32 bits	Current program status register
D0–D31	64 bits	Floating-point coprocessor registers for double precision
Q0–Q15	128 bits	Advanced SIMD registers
FPEXC	32 bits	Floating-point coprocessor, exception register
FPSCR	32 bits	Floating-point coprocessor, status and control register
FPSID	32 bits	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	Floating-point coprocessor registers for single precision
SPSR	32 bits	Saved program status register

*Table 8: Predefined register symbols*

In addition, specific cores might allow you to use other register symbols, for example APSR for the Cortex-M3, if available in the instruction syntax.

## PREDEFINED SYMBOLS

The 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.

These predefined symbols are available:

Symbol	Value
<code>__ARM_ADVANCED_SIMD__</code>	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor architecture has the Advanced SIMD architecture extension. The symbol is undefined for other cores.
<code>__ARM_MEDIA__</code>	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor architecture has the ARMv6 SIMD extension for multimedia. The symbol is undefined for other cores.
<code>__ARM_MPCORE__</code>	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor architecture has the Multiprocessing Extensions. The symbol is undefined for other cores.
<code>__ARM_PROFILE_M__</code>	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor is a profile M core. The symbol is undefined for other cores.
<code>__ARMVFP__</code>	An integer that is set based on the <code>--fpu</code> option and that identifies whether floating-point instructions for a vector floating-point coprocessor have been enabled or not. The symbol is defined to <code>__ARMVFPV2__</code> , <code>__ARMVFPV3__</code> , or <code>__ARMVFPV4__</code> . These symbolic names can be used when testing the <code>__ARMVFP__</code> symbol. 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. The build number does not necessarily increase with an assembler that is released later.
<code>__DATE__</code>	The current date in <code>dd/Mmm/yyyy</code> format (string).
<code>__FILE__</code>	The name of the current source file (string).
<code>__IAR_SYSTEMS_ASM__</code>	IAR assembler identifier (number). Note that the number could be higher in a future version of the product. This symbol can be tested with <code>#ifdef</code> to detect whether the code was assembled by an assembler from IAR Systems.
<code>__IASMARM__</code>	An integer that is set to 1 when the code is assembled with the IAR Assembler for ARM.
<code>__LINE__</code>	The current source line number (number).

Table 9: Predefined symbols

Symbol	Value
<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 6.21.2 is returned as 6021002 (number).

Table 9: Predefined symbols (Continued)

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

### Including symbol values in code

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

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

```

name      timeOfAssembly
extern   printStr
section  MYCODE:CODE(2)

adr      r0,time          ; Load address of time
                          ; string in R0.
bl       printStr        ; Call string output routine.
bx       lr               ; Return

data
time     dc8      __TIME__ ; In data mode:
                          ; String representing the
                          ; time of assembly.

end
```

### Testing symbols for conditional assembly

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

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

```
#if (__VER__ > 6021000)           ; New assembler version
;...
;...
#else                             ; Old assembler version
;...
;...
#endif
```

For more information, see *Conditional assembly directives*, page 80.

## 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 DATA and CODE as follows:

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

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

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

## EXPRESSION RESTRICTIONS

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

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

**No forward**

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

**No external**

No external references in the expression are allowed.

**Absolute**

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

**Fixed**

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

---

## List file format

The format of an assembler list file is as follows:

**HEADER**

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

**BODY**

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

- The line number in the source file. Lines generated by macros, if listed, have a . (period) in the source line number field.
- The address field shows the location in memory, which can be absolute or relative depending on the type of section. The notation is hexadecimal.
- The data field shows the data generated by the source line. The notation is hexadecimal. Unresolved values are represented by ..... (periods), where two periods signify one byte. These unresolved values are resolved during the linking process.
- The assembler source line.

## SUMMARY

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

## SYMBOL AND CROSS-REFERENCE TABLE

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

This information is provided for each symbol in the table:

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

*Table 10: Symbol and cross-reference table*

---

## Programming hints

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

### 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.

#### **Example**

The UART read address `0x40050000` of the device is defined in the `ionuc100.h` file as:

```
__IO_REG32_BIT(UA0_RBR, 0x40050000, __READ_WRITE, __uart_rbr_bits)
```

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

```
UA0_RBR DEFINE 0x40050000
```

## 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 102.

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





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

---

## Using command line assembler options

To set assembler options from the command line, include them 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 displays 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 this command to generate a list file to the default filename (`power2.lst`):

```
iasmarm power2.s -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.s -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.s -Llist\
```

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

This 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 preprocessor symbols
<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>	Floating-point coprocessor architecture configuration
<code>-G</code>	Opens standard input as source
<code>-g</code>	Disables the automatic search for system include files
<code>-I</code>	Adds a search path for a header file
<code>-i</code>	Lists <code>#included</code> text
<code>-j</code>	Enables alternative register names, mnemonics, and operators
<code>-L</code>	Generates a list file to path
<code>-l</code>	Generates a list file
<code>--legacy</code>	Generates code linkable with older toolchains.
<code>-M</code>	Macro quote characters
<code>-N</code>	Omits header from the assembler listing
<code>-n</code>	Enables support for multibyte characters
<code>-O</code>	Sets the object filename to path
<code>-o</code>	Sets the object filename

*Table 11: Assembler options summary*

Command line option	Description
-p	Sets the number of lines per page in the list file
-r	Generates debug information.
-S	Sets silent operation
-s	Case-sensitive user symbols
--system_include_dir	Specifies the path for system include files
-t	Tab spacing
-U	Undefines a symbol
-w	Disables warnings
-x	Includes cross-references

Table 11: Assembler options summary (Continued)

## Description of assembler options

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



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

### -B

Syntax

-B

Description

Use this option to make the assembler print macro execution information to the standard output stream for every call to 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`.

See also

`-L`, page 41.



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

**-c**

Syntax `-c {D|M|E|A|O}`

## Parameters

D	Disables list file
M	Includes macro definitions
E	Excludes macro expansions
A	Includes assembled lines only
O	Includes multiline code

## Description

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` or `-l`.

## See also

`-L`, page 41.



To set related options, select:

**Project>Options>Assembler >List**

**--cpu**

Syntax `--cpu target_core`

## Parameters

*target\_core* Can be values such as ARM7TDMI or architecture versions, for example 4T. ARM7TDMI is the default value.

## Description

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

## See also

The *IAR C/C++ Development Guide for ARM* for a complete list of coprocessor architecture variants.



**Project>Options>General Options>Target>Processor variant>Core**

**-D**

Syntax	<code>-Dsymbol [=value]</code>	
Parameters	<i>symbol</i>	The name of the symbol you want to define.
	<i>value</i>	The value of the symbol. If no value is specified, 1 is used.
Description	Use this option to define a symbol to be used by the preprocessor.	
Example	<p>You might want to arrange your source code to produce either the test version or the production version of your application, depending on whether the symbol <code>TESTVER</code> was defined. To do this, use include sections such as:</p>	

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

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

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

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

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



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

**-E**

Syntax	<code>-E<i>number</i></code>	
Parameters	<i>number</i>	The number of errors before the assembler stops the assembly. <i>number</i> must be a positive integer; 0 indicates no limit.
Description	Use this option to specify the maximum number of errors that the assembler reports. By default, the maximum number is 100.	



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

**-e**

Syntax `-e`

Description Use this option to cause 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**

Syntax `--endian={little|l|big|b}`

Parameters

<code>little, l (default)</code>	Specifies little-endian byte order.
<code>big, b</code>	Specifies big-endian byte order.

Description Use this option to specify the byte order of the generated code and data.



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

**-f**

Syntax `-f filename`

Parameters

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

For information about specifying a filename, see *Using command line assembler options*, page 33.

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

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

Example

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

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



To set this option, use:

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

## --fpu

Syntax

`--fpu fpu_variant`

Parameters

*fpu\_variant*

A floating-point coprocessor architecture variant, such as VFPv3 or none (default).

Description

Use this option to specify the floating-point coprocessor architecture variant and get the correct instruction set and registers.

See also

The *IAR C/C++ Development Guide for ARM* for a complete list of coprocessor architecture variants.



**Project>Options>General Options>Target>FPU**

## -G

Syntax

`-G`

Description

Use this option to make the assembler read the source from the standard input stream, rather than from a specified source file.

When `-G` is used, you cannot specify a source filename.



This option is not available in the IDE.

## -g

Syntax

`-g`

Description

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



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

**-I**

Syntax	<code>-Ipath</code>
Parameters	<code>path</code> The search path for #include files.
Description	<p>Use this option to specify paths to be used by the preprocessor. This option can be used more than once on the command line.</p> <p>By default, the assembler searches for #include files in the current working directory, in the system header directories, and in the paths specified in the <code>IASMARM_INC</code> environment variable. The <code>-I</code> option allows you to give the assembler the names of directories which it will also search if it fails to find the file in the current working directory.</p>
Example	<p>For example, using the options:</p> <pre>-Ic:\global\ -Ic:\thisproj\headers\</pre> <p>and then writing:</p> <pre>#include "asmlib.hdr"</pre> <p>in the source code, make the assembler search first in the current directory, then in the directory <code>c:\global\</code>, and then in the directory <code>C:\thisproj\headers\</code>. Finally, the assembler searches the directories specified in the <code>IASMARM_INC</code> environment variable, provided that this variable is set, and in the system header directories.</p>



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

**-i**

Syntax	<code>-i</code>
Description	<p>Use this option to list #include files in the list file.</p> <p>By default, the assembler does not list #include file lines because these often come from standard files and would waste space in the list file. The <code>-i</code> option allows you to list these file lines.</p>



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



**-j**

Syntax	-j
Description	Use this option to enable alternative register names, mnemonics, and operators in order to increase compatibility with other assemblers and allow porting of code.
See also	<i>Operator synonyms</i> , page 51 and the chapter <i>Migrating to the IAR Assembler for ARM</i> .



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

**-L**

Syntax	-L[ <i>path</i> ]				
Parameters	<table> <tr> <td>No parameter</td> <td>Generates a listing with the same name as the source file, but with the filename extension <code>lst</code>.</td> </tr> <tr> <td><i>path</i></td> <td>The path to the destination of the list file. Note that you must not include a space before the path.</td> </tr> </table>	No parameter	Generates a listing with the same name as the source file, but with the filename extension <code>lst</code> .	<i>path</i>	The path to the destination of the list file. Note that you must not include a space before the path.
No parameter	Generates a listing with the same name as the source file, but with the filename extension <code>lst</code> .				
<i>path</i>	The path to the destination of the list file. Note that you must not include a space before the path.				
Description	By default, the assembler does not generate a list file. Use this option to make the assembler generate one and send it to the file <code>[<i>path</i>]sourcefilename.lst</code> . -L cannot be used at the same time as -l.				

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

```
iasmarm prog -Llist\
```



To set related options, select:

**Project>Options>Assembler >List**

**-l**

Syntax	-l <i>filename</i>		
Parameters	<table> <tr> <td><i>filename</i></td> <td>The output is stored in the specified file. Note that you must include a space before the filename. If no extension is specified, <code>lst</code> is used.</td> </tr> </table>	<i>filename</i>	The output is stored in the specified file. Note that you must include a space before the filename. If no extension is specified, <code>lst</code> is used.
<i>filename</i>	The output is stored in the specified file. Note that you must include a space before the filename. If no extension is specified, <code>lst</code> is used.		

For information about specifying a filename, see *Using command line assembler options*, page 33.

## Description

Use this option to make the assembler generate a listing and send it to the file *filename*. By default, the assembler does not generate a list file.

To generate a list file with the default filename, use the `-L` option instead.



To set related options, select:

**Project>Options>Assembler >List**

## --legacy

## Syntax

```
--legacy={RVCT3.0}
```

## Parameters

RVCT3.0

Specifies the linker in RVCT3.0. Use this parameter together with the `--aeabi` option to generate code that should be linked with the linker in RVCT3.0.

## Description

Use this option to generate object code that is compatible with the specified toolchain.



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

## -M

## Syntax

```
-Mab
```

## Parameters

*ab*

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

## Description

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

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

## Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

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

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



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

## -N

Syntax

-N

Description

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 also

`-L`, page 41.



**Project>Options>Assembler >List>Include header**

## -n

Syntax

-n

Description

By default, multibyte characters cannot be used in assembler source code. Use this option to interpret multibyte characters in the source code according to the host computer's default setting for multibyte support.

Multibyte characters are allowed in C/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**

Syntax `-O[path]`

## Parameters

*path* The path to the destination of the object file. Note that you must not include a space before the path.

## Description

Use this option to set the path to be used on the name of the object file.

By default, the path is null, so the object filename corresponds to the source filename. The `-O` option lets you specify a path, for example, to direct the object file to a subdirectory.

Note that `-O` cannot 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**

Syntax `-o {filename|directory}`

## Parameters

*filename* The object code is stored in the specified file.

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

For information about specifying a filename or directory, see *Using command line assembler options*, page 33.

## Description

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

The `-o` option cannot be used at the same time as the `-O` option.



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

**-p**

Syntax `-plines`

## Parameters

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

## Description

Use this option to set the number of lines per page explicitly.  
This option is used in conjunction with the list options `-L` or `-l`.

## See also

`-L`, page 41.



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

**-r**

Syntax `-r`

## Description

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



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

**-S**

Syntax `-S`

## Description

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

The assembler sends error and warning messages to the error output stream, so they are displayed regardless of this setting.



This option is not available in the IDE.

**-s**

Syntax	<code>-s {+ -}</code>				
Parameters	<table> <tr> <td>+</td> <td>Case-sensitive user symbols.</td> </tr> <tr> <td>-</td> <td>Case-insensitive user symbols.</td> </tr> </table>	+	Case-sensitive user symbols.	-	Case-insensitive user symbols.
+	Case-sensitive user symbols.				
-	Case-insensitive user symbols.				
Description	Use this option to control whether the assembler is sensitive to the case of user symbols. By default, case sensitivity is on.				
Example	By default, for example <code>LABEL</code> and <code>label</code> refer to different symbols. When <code>-s</code> is used, <code>LABEL</code> and <code>label</code> instead refer to the same symbol.				



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

**--system\_include\_dir**

Syntax	<code>--system_include_dir path</code>		
Parameters	<table> <tr> <td><i>path</i></td> <td>The path to the system include files.</td> </tr> </table> <p>For information about specifying a filename or directory, see <i>Using command line assembler options</i>, page 33.</p>	<i>path</i>	The path to the system include files.
<i>path</i>	The path to the system include files.		
Description	By default, the assembler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location.		



This option is not available in the IDE.

**-t**

Syntax	<code>-tn</code>		
Parameters	<table> <tr> <td><i>n</i></td> <td>The tab spacing; must be in the range 2 to 9.</td> </tr> </table>	<i>n</i>	The tab spacing; must be in the range 2 to 9.
<i>n</i>	The tab spacing; must be in the range 2 to 9.		
Description	By default, the assembler sets 8 character positions per tab stop. Use this option to specify a different tab spacing.		

This option is useful in conjunction with the list options `-L` or `-l`.

See also

`-L`, page 41.



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

## -U

Syntax

`-Usymbol`

Parameters

*symbol*

The predefined symbol to be undefined.

Description

By default, the assembler provides certain predefined symbols.

Use this option 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__
```

See also

*Predefined symbols*, page 25.



This option is not available in the IDE.


## -w

Syntax


`-w[+|-|+n|-n|+m-n|-m-n][s]`

Parameters

No parameter	Disables all warnings.
<code>+</code>	Enables all warnings.
<code>-</code>	Disables all warnings.
<code>+<i>n</i></code>	Enables just warning <i>n</i> .
<code>-<i>n</i></code>	Disables just warning <i>n</i> .
<code>+<i>m-n</i></code>	Enables warnings <i>m</i> to <i>n</i> .
<code>-<i>m-n</i></code>	Disables warnings <i>m</i> to <i>n</i> .

	<code>s</code>	Generates the exit code 1 if a warning message is produced. By default, warnings generate exit code 0.
Description		By default, the assembler displays a warning message when it detects an element of the source code which is legal in a syntactical sense, but might contain a programming error. Use this option to disable all warnings, a single warning, or a range of warnings. Note that the <code>-w</code> option can only be used once on the command line.
Example		To disable just warning 0 (unreferenced label), use this command: <code>iasmarm prog -w-0</code> To disable warnings 0 to 8, use this command: <code>iasmarm prog -w-0-8</code>
See also		<i>Assembler diagnostics</i> , page 131. To set related options, select: <b>Project&gt;Options&gt;Assembler&gt;Diagnostics</b>

**-X**

Syntax	<code>-x{D I 2}</code>							
Parameters		<table> <tr> <td>D</td> <td>Includes preprocessor <code>#defines</code>.</td> </tr> <tr> <td>I</td> <td>Includes internal symbols.</td> </tr> <tr> <td>2</td> <td>Includes dual-line spacing.</td> </tr> </table>	D	Includes preprocessor <code>#defines</code> .	I	Includes internal symbols.	2	Includes dual-line spacing.
D	Includes preprocessor <code>#defines</code> .							
I	Includes internal symbols.							
2	Includes dual-line spacing.							
Description		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 <code>-L</code> or <code>-l</code> .						
See also		<code>-L</code> , page 41. <b>Project&gt;Options&gt;Assembler&gt;List&gt;Include cross reference</b>						



# 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 assembler operators

Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The precedence numbers range from 1 (the highest precedence, that is, first evaluated) to 7 (the lowest precedence, that is, last evaluated).

These rules determine how expressions are evaluated:

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

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

---

## Summary of assembler operators

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

### PARENTHESIS OPERATOR – I

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

### UNARY OPERATORS – I

+	Unary plus.
-	Unary minus.
!	Logical NOT.

~	Bitwise NOT.
LOW	Low byte.
HIGH	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.
SIZEOF	Section size.

## **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

=, == Equal.

<>, != Not equal.

> Greater than.

< Less than.

UGT Unsigned greater than.

ULT Unsigned less than.

>= Greater than or equal.

<= Less than or equal.

## OPERATOR SYNONYMS

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

Operator synonym	Precedence	Operator	Precedence	Function
:AND:	3	&	5	Bitwise AND.
:EOR:	3	^	6	Bitwise exclusive OR.
:LAND:	8	&&	5	Logical AND.
:LEOR:	8	XOR	6	Logical exclusive OR.
:LNOT:	1	!	1	Logical NOT.
:LOR:	6		6	Logical OR.
:MOD:	2	%	2	Modulo.

Table 12: Operator synonyms

Operator synonym	Precedence	Operator	Precedence	Function
:NOT:	1	~	1	Bitwise NOT.
:OR:	3		6	Bitwise OR.
:SHL:	2.5	<<	4	Logical shift left.
:SHR:	2.5	>>	4	Logical shift right.

Table 12: Operator synonyms

**Note:** The operator synonyms are enabled by the option `-j`. In some cases, the ARM operators and the operator synonyms have different precedences. See also the chapter *Migrating to the IAR Assembler for ARM*.

## Description of assembler operators

This section gives detailed descriptions of each assembler operator. The number within parentheses specifies the precedence of the operator.

For related information, see *Expressions, operands, and operators*, page 22.

### ( ) Parenthesis (1)

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

**Example**  
 $1+2*3 \rightarrow 7$   
 $(1+2)*3 \rightarrow 9$

### \* Multiplication (2)

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

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

## + Unary plus (1)

Description                      Unary plus operator.

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

## + Addition (3)

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

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

## - Unary minus (1)

Description                      The unary minus operator performs arithmetic negation on its operand.

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

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

## - Subtraction (3)

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

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

## / Division (2)

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

Example	9/2 → 4 -12/3 → -4 9/2*6 → 24
---------	-------------------------------------

**< Less than (7)**

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

Example	-1 < 2 → 1 2 < 1 → 0 2 < 2 → 0
---------	--------------------------------------

**<= Less than or equal (7)**

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

Example	1 <= 2 → 1 2 <= 1 → 0 1 <= 1 → 1
---------	--

**<>, != Not equal (7)**

Description	<> 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)**

Description	= 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)

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

Example

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

## >= Greater than or equal (7)

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

Example

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

## && Logical AND (5)

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

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

Example

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

## & Bitwise AND (5)

Description Use `&` or the synonym `:AND:` 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

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

**~ Bitwise NOT (1)**

**Description** Use `~` or the synonym `:NOT:` 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** `~ 1010B → 1111111111111111111111111111111110101B`

**| Bitwise OR (6)**

**Description** Use `|` or the synonym `:OR:` 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** `1010B | 0101B → 1111B`  
`1010B | 0000B → 1010B`

**^ Bitwise exclusive OR (6)**

**Description** Use `^` or the synonym `:EOR:` 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** `1010B ^ 0101B → 1111B`  
`1010B ^ 0011B → 1001B`

**% Modulo (2)**

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

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

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



## ! Logical NOT (1)

Description Use `!` or the synonym `:LNOT`: to negate a logical argument.

Example `! 0101B → 0`  
`! 0000B → 1`

## || Logical OR (6)

Description Use `||` or the synonym `:LOR`: to perform a logical OR between two integer operands.

Example `1010B || 0000B → 1`  
`0000B || 0000B → 0`

## << Logical shift left (4)

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

**Note:** The precedence of `:SHL`: is 2.5.

Example `00011100B << 3 → 11100000B`  
`00000111111111111111B << 5 → 1111111111100000B`  
`14 << 1 → 28`

## >> Logical shift right (4)

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

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

Example `01110000B >> 3 → 00001110B`  
`11111111111111111111B >> 20 → 0`  
`14 >> 1 → 7`

**BYTE1 First byte (I)**

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

Example `BYTE1 0xABCD → 0xCD`

**BYTE2 Second byte (I)**

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

Example `BYTE2 0x12345678 → 0x56`

**BYTE3 Third byte (I)**

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

Example `BYTE3 0x12345678 → 0x34`

**BYTE4 Fourth byte (I)**

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

Example `BYTE4 0x12345678 → 0x12`

**DATE Current time/date (I)**

Description 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 (I)

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

Example            HIGH 0xABCD → 0xAB

## HWRD High word (I)

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

Example            HWRD 0x12345678 → 0x1234

## LOW Low byte (I)

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

Example            LOW 0xABCD → 0xCD

## LWRD Low word (I)

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

Example            LWRD 0x12345678 → 0x5678

## SFB Section begin (I)

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

## Parameters

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

## Description

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

## Example

```

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

```

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

## SFE Section end (I)

## Syntax

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

## Parameters

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

## Description

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

## Example

```

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

```

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

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

## SIZEOF Section size (I)

Syntax	<code>SIZEOF section</code>
Parameters	<p><code>section</code>                      The name of a relocatable section, which must be defined before <code>SIZEOF</code> is used.</p>
Description	<code>SIZEOF</code> generates <code>SFE-SFB</code> for its argument. That is, it calculates the size in bytes of a section. This is done when modules are linked together.
Example	<p>These two files set <code>size</code> to the size of the section <code>MYCODE</code>.</p> <p>Table.s:</p> <pre> module table section MYCODE:CODE ; Forward declaration of MYCODE. section SEGTAB:CONST(2) data size dc32 sizeof(MYCODE) end </pre> <p>Application.s:</p> <pre> module application section MYCODE:CODE(2) nop ; Placeholder for application. end </pre>

## UGT Unsigned greater than (7)

Description	<code>UGT</code> evaluates to 1 (true) if the left operand has a larger value than the right operand, otherwise it is 0 (false). The operation treats the operands as unsigned values.
Example	<pre> 2 UGT 1 → 1 -1 UGT 1 → 1 </pre>

## ULT Unsigned less than (7)

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

Example                    1 ULT 2 → 1  
                             -1 ULT 2 → 0

## **XOR Logical exclusive OR (6)**

Description                XOR or the synonym :LEOR: 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                    0101B XOR 1010B → 0  
                             0101B XOR 0000B → 1

# Assembler directives

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

---

## Summary of assembler directives

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

- *Module control directives*, page 67
- *Symbol control directives*, page 70
- *Mode control directives*, page 72
- *Section control directives*, page 74
- *Value assignment directives*, page 78
- *Conditional assembly directives*, page 80
- *Macro processing directives*, page 82
- *Listing control directives*, page 90
- *C-style preprocessor directives*, page 94
- *Data definition or allocation directives*, page 99
- *Assembler control directives*, page 102
- *Call frame information directives*, page 105.

This table gives a summary of all the assembler directives:

<b>Directive</b>	<b>Description</b>	<b>Section</b>
<code>_args</code>	Is set to number of arguments passed to macro.	Macro processing
<code>\$</code>	Includes a file.	Assembler control
<code>#define</code>	Assigns a value to a label.	C-style preprocessor
<code>#elif</code>	Introduces a new condition in an <code>#if...#endif</code> block.	C-style preprocessor
<code>#else</code>	Assembles instructions if a condition is false.	C-style preprocessor
<code>#endif</code>	Ends an <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> block.	C-style preprocessor
<code>#error</code>	Generates an error.	C-style preprocessor
<code>#if</code>	Assembles instructions if a condition is true.	C-style preprocessor
<code>#ifdef</code>	Assembles instructions if a symbol is defined.	C-style preprocessor

*Table 13: Assembler directives summary*

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

---

Table 13: Assembler directives summary (Continued)



Directive	Description	Section
DC16	Generates 16-bit constants.	Data definition or allocation
DC24	Generates 24-bit constants.	Data definition or allocation
DC32	Generates 32-bit constants.	Data definition or allocation
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	Ends the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing
ENDR	Ends a repeat structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment

Table 13: Assembler directives summary (Continued)

<b>Directive</b>	<b>Description</b>	<b>Section</b>
EVEN	Aligns the program counter to an even address.	Section control
EXITM	Exits prematurely from a macro.	Macro processing
EXTERN	Imports an external symbol.	Symbol control
EXTWEAK	Imports an external symbol; the symbol may be undefined.	Symbol control
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
LSTSAS	Controls structured assembly listing.	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

Table 13: Assembler directives summary (Continued)

Directive	Description	Section
PROGRAM	Begins a module.	Module control
PUBLIC	Exports symbols to other modules.	Symbol control
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.	Symbol control
RADIX	Sets the default base.	Assembler control
REPT	Assembles instructions a specified number of times.	Macro processing
REPTC	Repeats and substitutes characters.	Macro processing
REPTI	Repeats and substitutes strings.	Macro processing
REQUIRE	Forces a symbol to be referenced.	Symbol control
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 13: 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. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 28.

Directive	Description	Expression restrictions
AAPCS	Sets module attributes that informs the linker that all exported functions in the module follows the Procedure Call Standard for the ARM Architecture, AAPCS.	The assembler does not verify that the claims are fulfilled.
END	Ends the assembly of the last module in a file.	Locally defined symbols plus offset or integer constants

Table 14: Module control directives

Directive	Description	Expression restrictions
NAME	Begins a module; alias to PROGRAM.	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.	
RTMODEL	Declares runtime model attributes.	Not applicable

Table 14: Module control directives (Continued)

## SYNTAX

AAPCS [*modifier* [...]]

END

NAME *symbol*

PRESERVE8

PROGRAM *symbol*

REQUIRE8

RTMODEL *key, value*

## PARAMETERS

*key* A text string specifying the key.

*modifier* An AAPCS extension; possible values are INTERWORK, VFP, VFP\_COMPATIBLE, ROPI, RWPI, RWPI\_COMPATIBLE. Modifiers can be combined to specify AAPCS variants.

*symbol* Name assigned to module.

*value* 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 *IAR C/C++ Development Guide for ARM*.

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

### Terminating the source file

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

### 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`, optionally with modifiers, 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 *IAR C/C++ Development Guide for ARM*.

### Examples

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

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

- MOD\_2 and MOD\_3 *can* be linked together since they have no runtime model conflicts. The value \* matches any runtime model value.

Assembler source file f1.s:

```
module mod_1
rtmodel "CAN",      "ISO11519"
rtmodel "Platform", "M7"
; ...
end
```

Assembler source file f2.s:

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

Assembler source file f3.s:

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

---

## Symbol control directives

These directives control how symbols are shared between modules:

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

Table 15: Symbol control directives

### SYNTAX

```
EXTERN symbol [, symbol] ...
```

```
EXTWEAK symbol [, symbol] ...
```

```
IMPORT symbol [, symbol] ...
```

```
PUBLIC symbol [, symbol] ...
PUBWEAK symbol [, symbol] ...
REQUIRE symbol
```

## PARAMETERS

*symbol*                      Symbol to be imported or exported.

## DESCRIPTIONS

### Exporting symbols to other modules

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

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

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

### Exporting symbols with multiple definitions to other modules

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

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

### Importing symbols

Use `EXTERN` or `IMPORT` to import an untyped external symbol.

The `REQUIRE` directive marks a symbol as referenced. This is useful if the section containing the symbol must be loaded even if the code is not referenced.

## EXAMPLES

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

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

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

```

                name    errorMessage
                extern  print
                public  err

                section MYCODE:CODE(2)
                arm

err            adr     r0,msg
                bl     print
                bx     lr

                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 16: 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.

The use of the mode directives `CODE32` and `CODE16` is deprecated. Instead, use `ARM` and `THUMB`, respectively.

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 IAR Assembler for ARM* for details.

The assembler will initially be in `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:

```

        name    modeChange
        section MYCODE:CODE(2)
        thumb
thumbEntry
        bx      pc          ; Branch to armEntry, and
                           ; change execution mode.
        nop      ; For alignment only.
        arm
armEntry
        ; ...

        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 THUMB section.

```

                name      dataDirective
                section MYCODE:CODE(2)
                thumb
thumbLabel    ldr      r0,dataLabel
                bx      lr

                data                                ; Change to data mode, so
                                                    ; that bit 0 is not set
                                                    ; on labels.

dataLabel    dc32    0x12345678
                dc32    0x12345678

                end

```

---

## Section control directives

The section directives control how code and data are located. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 28.

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 an ELF section; alias to SECTION.	No external references Absolute
SECTION	Begins an ELF section.	No external references Absolute
SECTION_TYPE	Sets ELF type and flags for a section.	

Table 17: Section control directives

## SYNTAX

```

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

```

## PARAMETERS

<i>align</i>	The power of two to which the address should be aligned. The permitted range is 0 to 8. The default align value is 0, except for code sections where the default is 1.
<i>flag</i>	<p>ROOT, NOROOT</p> <p>ROOT (the default mode) indicates that the section fragment must not be discarded.</p> <p>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.</p> <p>REORDER, NOREORDER</p> <p>NOREORDER (the default mode) starts a new fragment in the section with the given name, or a new section if no such section exists.</p> <p>REORDER starts a new section with the given name.</p>
<i>section</i>	The name of the section. The section name is a user-defined symbol that follows the rules described in <i>Symbols</i> , page 24.
<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 having to save the current program location counter.

**Note:** The first instance of a `SECTION` or `RSEG` directive must not be preceded by any code generating directives, such as `DC8` or `DS8`, or by any assembler instructions.

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

### Aligning a section

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

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

`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.

For both RAM and ROM, the parameter `align` can be within the range 0 to 30.

## EXAMPLES

### Beginning a relocatable section

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

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

```

                                name    calculate
                                extern  subrtn,divrtn

                                section MYDATA:DATA (2)
                                data
funcTable  dc32    subrtn
                                dc32    divrtn

                                section MYCODE:CODE (2)
                                arm
main       ldr     r0,=funcTable   ; Get address, and
                                ldr     pc,[r0]       ; jump to it.
                                end

```

### Aligning a section

This example starts a section and adds some data. It then aligns to a 64-byte boundary before creating a 64-byte table. The section has an alignment of 64 bytes to ensure the 64-byte alignment of the table.

```

                                name    alignment
                                section MYDATA:DATA(6) ; Start a relocatable data
                                                                ; section 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, and
target2   ds16    1                ; two more bytes of data.
                                alignram 3                ; Align to an 8-byte boundary
ages      ds8     64               ; and 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 18: Value assignment directives

### SYNTAX

```

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

```

### PARAMETERS

<i>const_expr</i>	Constant value assigned to symbol.
<i>expr</i>	Value assigned to symbol or value to be tested.
<i>label</i>	Symbol to be defined.

### DESCRIPTIONS

#### Defining a temporary value

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

#### Defining a permanent local value

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

Use `EXTERN` to import symbols from other modules.

### Defining a permanent global value

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

A symbol which was given a value with `DEFINE` can be made available to modules in other files with the `PUBLIC` directive.

Symbols defined with `DEFINE` cannot be redefined within the same file. Also, the expression assigned to the defined symbol must be constant.

## EXAMPLES

### Redefining a symbol

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

```

                                name    table
cons                            set      1

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

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

```

It generates this code:

```

9
10                                name    table
10                                cons    set      1
11
12                                ; Generate table of powers of 3.
20
21                                section .text:CODE(2)
22                                table    cr_tabl 4
22 00000000 01000000            table    cr_tabl 4
22.1                                cons    set      cons * 3
22.2                                if      4 > 1
22                                cr_tabl 4 - 1

```

```

22      00000004 03000000      table      cr_tabl 4
22.1                                cons        set      cons * 3
22.2                                if          4 - 1 > 1
22      cr_tabl 4 - 1 - 1
22      00000008 09000000      table      cr_tabl 4
22.1                                cons        set      cons * 3
22.2                                if          4 - 1 - 1 > 1
22      cr_tabl 4 - 1 - 1 - 1
22      0000000C 1B000000      table      cr_tabl 4
22.1                                cons        set      cons * 3
22.2                                if          4 - 1 - 1 - 1 > 1
22.3                                cr_tabl 4 - 1 - 1 - 1 - 1
22.4                                endif
22.5                                endm
22.6                                endif
22.7                                endm
22.8                                endif
22.9                                endm
22.10                               endif
22.11                               endm
23      end

```

---

## Conditional assembly directives

These directives provide logical control over the selective assembly of source code. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 28.

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

Table 19: Conditional assembly directives



## SYNTAX

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

## PARAMETERS

<i>condition</i>	One of these:	
	An absolute expression	The expression must not contain forward or external references, and any non-zero value is considered as true.
	<i>string1=string2</i>	The condition is true if <i>string1</i> and <i>string2</i> have the same length and contents.
	<i>string1&lt;&gt;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 ENDF directives to control the assembly process at assembly time. If the condition following the IF directive is not true, the subsequent instructions do not generate any code (that is, it is not assembled or syntax checked) until an ELSE or ENDF directive is found.

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

All assembler directives (except for END) as well as the inclusion of files can be disabled by the conditional directives. Each IF directive must be terminated by an ENDF directive. The ELSE directive is optional, and if used, it must be inside an IF . . . ENDF block. IF . . . ENDF and IF . . . ELSE . . . ENDF blocks can be nested to any level.

## EXAMPLES

This example uses a macro to add a constant to a register:

```
?add      macro   a,b,c
           if     _args == 2
           adds  a,a,#b
```

```

        elseif _args == 3
        adds   a,b,#c
        endif
        endm

        name   addWithMacro
        section MYCODE:CODE(2)
        arm

main    ?add   r1,0xFF      ; This,
        ?add   r1,r1,0xFF  ; and this,
        adds   r1,r1,#0xFF ; are the same as this.

        end

```

## Macro processing directives

These directives allow user macros to be defined. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 28.

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

Table 20: Macro processing directives

### SYNTAX

```

_args
ENDM
ENDR
EXITM

```

```

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

```

## PARAMETERS

<i>actual</i>	Strings to be substituted.
<i>argument</i>	Symbolic argument names.
<i>expr</i>	An expression.
<i>formal</i>	An argument into which each character of <i>actual</i> (REPTC) or each string of <i>actual</i> (REPTI) is substituted.
<i>name</i>	The name of the macro.
<i>symbol</i>	Symbols 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 `errMacro` as follows:

```

                                name    errMacro
                                extern  abort
errMac                          macro   text
                                bl      abort

```

```

data
dc8      text,0
endm

```

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

```

section MYCODE:CODE(2)
arm
errMac  'Disk not ready'

```

The assembler expands this to:

```

section MYCODE:CODE(2)
arm
bl      abort
data
dc8     'Disk not ready',0

end

```

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

The previous example could therefore be written as follows:

```

errMac  name    errMacro
        extern  abort
        macro
        bl      abort
        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:

```

                                name    cmpMacro
cmp_reg      macro    op
                                CMP    op
                                endm

```

The macro can be called using the macro quote characters:

```

                                section MYCODE:CODE(2)
                                cmp_reg <r3,r4>
                                end

```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 42.

### Predefined macro symbols

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

```

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

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

```

It generates this code:

```

19
20
21
21.1
21.2
21.3
21.4
21.5
21    00000000 03
21.1
21.2

                                module  filler
                                section .text:CODE(2)
                                fill    3
                                if      _args == 2
                                rept
                                dc8     3
                                endr
                                else
                                fill    3
                                endif
                                endm

```

```

22                               fill    4, 3
22.1                             if      _args == 2
22.2                             rept    3
22.3                             dc8    4
22.4                             endr
22    00000001 04                dc8    4
22    00000002 04                dc8    4
22    00000003 04                dc8    4
22.1                             else
22.2                             dc8    4
22.3                             endif
22.4                             endm
23                               end

```

## How macros are processed

The macro process consists of three distinct phases:

- 1 The assembler scans and saves macro definitions. The text between `MACRO` and `ENDM` is saved but not syntax checked. Include-file references `$file` are recorded and 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 continues to be the output from the macro processor, until all lines of the current macro definition have been read.

## Repeating statements

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

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

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

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

## EXAMPLES

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

### Coding inline for efficiency

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

This example outputs bytes from a buffer to a port:

```

                                name    ioBufferSubroutine
                                section MYCODE:CODE(2)
                                arm
play                               ldr    r1,=buffer      ; Pointer to buffer.
                                ldr    r2,=ioPort      ; Pointer to ioPort.
                                ldr    r3,=512        ; Size of buffer.
                                add    r3,r3,r1        ; Address of first byte
                                ; after buffer.
loop                               ldrb   r4,[r1],#1      ; Read a byte of data, and
                                strb   r4,[r2]        ; write it to the ioPort.
                                cmp    r1, r3        ; Reached first byte after?
                                bne    loop          ; No: repeat.
                                bx     lr            ; Return.

ioPort                             equ    0x0100

                                section MYDATA:DATA(2)
                                data
buffer                             ds8    512          ; Reserve 512 bytes.

                                section MYCODE:CODE(2)
                                arm
main                               bl     play
done                              b     done

                                end

```

For efficiency we can recode this using a macro:

```

play                               name    ioBufferInline
                                macro   buf,size,port
                                local   loop
                                ldr    r1,=buf        ; Pointer to buffer.
                                ldr    r2,=port      ; Pointer to ioPort.

```

```

                                ldr    r3,=size      ; Size of buffer.
                                add    r3,r3,r1      ; Address of first byte
                                                ; after buffer.
loop    ldrb   r4,[r1],#1      ; Read a byte of data, and
                                strb   r4,[r2]      ; write it to the ioPort.
                                cmp    r1, r3      ; Reached first byte after?
                                bne    loop         ; No: repeat.
                                endm

ioPort  equ    0x0100

                                section MYDATA:DATA(2)
                                data
buffer  ds8    512              ; Reserve 512 bytes.

                                section MYCODE:CODE(2)
                                arm
main    play   buffer,512,ioPort
done    b      done

                                end

```

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

## Using REPTC and REPTI

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

```

                                name    reptc
                                extern  plotc
                                section MYCODE:CODE(2)

banner  reptc   chr,"Welcome"
                                movs   r0,#'chr'    ; Pass char as parameter.
                                bl      plotc
                                endr

                                end

```

This produces this code:

```

9                                name    reptc
10                               extern  plotc
11                               section MYCODE:CODE(2)
12
13                               banner  reptc   chr,"Welcome"

```



```

14                movs    r0,#'chr'      ; Pass char as
parameter.
15                bl      plotc
16                endr
16.1 00000000 5700B0E3  movs    r0,#'W'      ; Pass char as
parameter.
16.2 00000004 .....   bl      plotc
16.3 00000008 6500B0E3  movs    r0,#'e'      ; Pass char as
parameter.
16.4 0000000C .....   bl      plotc
16.5 00000010 6C00B0E3  movs    r0,#'l'      ; Pass char as
parameter.
16.6 00000014 .....   bl      plotc
16.7 00000018 6300B0E3  movs    r0,#'c'      ; Pass char as
parameter.
16.8 0000001C .....   bl      plotc
16.9 00000020 6F00B0E3  movs    r0,#'o'      ; Pass char as
parameter.
16.10 00000024 .....  bl      plotc
16.11 00000028 6D00B0E3  movs    r0,#'m'      ; Pass char as
parameter.
16.12 0000002C .....  bl      plotc
16.13 00000030 6500B0E3  movs    r0,#'e'      ; Pass char as
parameter.
16.14 00000034 .....  bl      plotc
17
18                end

```

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

```

                name     repti
                extern   a,b,c
                section  MYCODE:CODE(2)

clearABC       movs     r0,#0
                repti   location,a,b,c
                ldr     r1,=location
                str     r0,[r1]
                endr

                end

```

This produces this code:

```

9                name     repti
10               extern   a,b,c
11               section  MYCODE:CODE(2)
12
13 00000000 0000B0E3 clearABC  movs     r0,#0

```

```

14             loop      repti  location,a,b,c
15             ldr       r1,=location
16             str       r0,[r1]
17             endr
17.1 00000004 10109FE5  ldr       r1,=a
17.2 00000008 000081E5  str       r0,[r1]
17.3 0000000C 0C109FE5  ldr       r1,=b
17.4 00000010 000081E5  str       r0,[r1]
17.5 00000014 08109FE5  ldr       r1,=c
17.6 00000018 000081E5  str       r0,[r1]
18
19             end

```

## 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 21: Listing control directives

### SYNTAX

COL *columns*

LSTCND{+|-}

LSTCOD{+|-}

LSTEXP{+|-}

LSTMAC{+|-}

LSTOUT{+|-}

LSTPAG{+|-}

```
LSTREP{+|-}
LSTXRF{+|-}
PAGE
PAGSIZ 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 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-
; This section has already been debugged.
lstout+
; This section is currently being debugged.
end
```

### Listing conditional code and strings

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

```
name    lstcndTest
extern  print
section FLASH:CODE(2)

debug   set    0
begin   if    debug
        bl    print
        endif

        lstcnd+
begin2  if    debug
        bl    print
```

```
endif
```

```
end
```

This generates the following listing:

```

9          name    lstcndTest
10         extern  print
11         section FLASH:CODE(2)
12
13         debug   set    0
14         begin   if     debug
15         bl      print
16         endif
17
18         lstcnd+
19         begin2  if     debug
21         endif
22
23         end
```

## Controlling the listing of macros

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

```

          name    lstmacTest
          extern  memLoc
          section FLASH:CODE(2)

dec2     macro   arg
          subs   r1,r1,#arg
          subs   r1,r1,#arg
          endm

          lstmac+
inc2     macro   arg
          adds   r1,r1,#arg
          adds   r1,r1,#arg
          endm

begin    dec2    memLoc
          lstexp-
          inc2   memLoc
          bx     lr

; Restore default values for
; listing control directives.

          lstmac-
```

```
lstexp+
```

```
end
```

This produces the following output:

```

13                                     name    lstmacTest
14                                     extern  memLoc
15                                     section FLASH:CODE(2)
16
21
22                                     lstmac+
23             inc2                      macro   arg
24                                     adds    r1,r1,#arg
25                                     adds    r1,r1,#arg
26                                     endm
27
28             begin                      dec2    memLoc
28.1  00000000 .....                    subs    r1,r1,#memLoc
28.2  00000004 .....                    subs    r1,r1,#memLoc
28.3                                     endm
29                                     lstexp-
30                                     inc2    memLoc
31  00000010 1EFF2FE1                    bx      lr
32
33                                     ; Restore default values for
34                                     ; listing control directives.
35
36                                     lstmac-
37                                     lstexp+
38
39                                     end

```

---

## C-style preprocessor directives

The assembler has a C-style preprocessor that is similar to the C89 standard.

These C-language preprocessor directives are available:

Directive	Description
<code>#define</code>	Assigns a value to a preprocessor symbol.
<code>#elif</code>	Introduces a new condition in an <code>#if...#endif</code> block.
<code>#else</code>	Assembles instructions if a condition is false.
<code>#endif</code>	Ends an <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> block.

Table 22: C-style preprocessor directives

Directive	Description
<code>#error</code>	Generates an error.
<code>#if</code>	Assembles instructions if a condition is true.
<code>#ifdef</code>	Assembles instructions if a preprocessor symbol is defined.
<code>#ifndef</code>	Assembles instructions if a preprocessor symbol is undefined.
<code>#include</code>	Includes a file.
<code>#line</code>	Changes the source references in the debug information.
<code>#message</code>	Generates a message on standard output.
<code>#pragma</code>	This directive is recognized but ignored.
<code>#undef</code>	Undefines a preprocessor symbol.

Table 22: C-style preprocessor directives (Continued)

## SYNTAX

```
#define symbol text
#elif condition
#else
#endif
#error "message"
#if condition
#ifdef symbol
#ifndef symbol
#include {"filename" | <filename>}
#line line-no {"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 or referred.	
<i>line-no</i>	Source line number.	
<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

You must not mix assembler language and C-style preprocessor directives. Conceptually, they are different languages and mixing them might lead to unexpected behavior because an assembler directive is not necessarily accepted as a part of the C preprocessor language.

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

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

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

## Defining and undefining preprocessor symbols

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

```
#define symbol value
```

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

## Conditional preprocessor directives

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

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

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

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

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



## Including source files

Use `#include` to insert the contents of a header file into the source file at a specified point.

`#include "filename"` and `#include <filename>` search these directories in the specified order:

- 1 The source file directory. (This step is only valid for `#include "filename".`)
- 2 The directories specified by the `-I` option, or options. The directories are searched in the same order as specified on the command line, followed by the ones specified by environment variables.
- 3 The current directory, which is the same as where the assembler executable file is located.
- 4 The automatically set up library system include directories. See `-g`, page 39.

## 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.

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

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

```

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

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

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

                module misplacedComment2
                section MYCONST:CONST(2)

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

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

                end
```

### Changing the source line numbers

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

## EXAMPLES

### Using conditional preprocessor directives

This example defines the labels `tweak` and `adjust`. If `tweak` is defined, then register `r0` is decremented by an amount that depends on `adjust`, for example 30 when `adjust` is 3.

```
                name    calibrate
                extern  calibrationConstant
                section MYCODE:CODE(2)
                arm

#define    tweak 1
#define    adjust 3

calibrate    ldr    r0,calibrationConstant
#ifdef    tweak
#if    adjust==1
                subs    r0,r0,#4
#elif    adjust==2
                subs    r0,r0,#20
#elif    adjust==3
```

```

                                subs    r0,r0,#30
#endif
#endif    /* ifdef tweak */
                                str     r0,calibrationConstant
                                bx      lr

                                end

```

### Including a source file

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

```

; Exchange registers a and b.
; Use register c for temporary storage.

```

```

xch      macro    a,b,c
          movs    c,a
          movs    a,b
          movs    b,c
          endm

```

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

```

                                name    includeFile
                                section MYCODE:CODE(2)
                                arm

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

xchRegs  xch      r0,r1,r2
          bx      lr

                                end

```

---

## 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. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 28.

Directive	Alias	Description
DC8	DCB	Generates 8-bit constants, including strings.

Table 23: Data definition or allocation directives

<b>Directive</b>	<b>Alias</b>	<b>Description</b>
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 23: Data definition or allocation directives (Continued)

**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] ...
DS   count
DS8  count
DS16 count
DS24 count
DS32 count

```

**PARAMETERS**

*count*      A valid absolute expression specifying the number of elements to be reserved.

*expr*        A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings are zero filled to a multiple of the data size implied by the directive. Double-quoted strings are zero-terminated.

*value*       A valid absolute expression or floating-point constant.

## 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

This example sums up the entries of a constant table of 8-bit data.

```

                                module  sumTableAndIndex
                                section MYDATA:CONST
                                data

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

                                section MYCODE:CODE(2)
                                arm
count    set     0

addTable movs    r0,#0
          ldr     r1,=table

          rept   7
          if     count == 7
          exitm
          endif
          ldrb   r2,[r1,#count]
          adds   r0,r0,r2
count    set     count + 1
          endr

          bx     lr

          end

```

## Defining strings

To define a string:

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

To define a string which includes a trailing zero:

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

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

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

## Reserving space

To reserve space for 10 bytes:

```
table   DS8    10
```

---

## Assembler control directives

These directives provide control over the operation of the assembler. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 28.

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 24: 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. This is an alias for #include, see *Including source files*, page 97.

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 where the current literal pool is to be assembled. By default, this is performed at every END and RSEG directive. For an example, see *LDR (ARM)*, page 125.

### 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

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

```
; Exchange registers a and b.
; Use register c for temporary storage.

xch      macro    a,b,c
          movs    c,a
          movs    a,b
          movs    b,c
          endm
```

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

```
          name    includeFile
          section MYCODE:CODE(2)
          arm

; Standard macro definitions.
$Macros.inc

xchRegs  xch     r0,r1,r2
          bx     lr

          end
```

### Defining comments

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

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

See also *Comments in C-style preprocessor directives*, page 97.

### Changing the base

To set the default base to 16:

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

          radix  16           ; With the default base set
```



```

movs    r0,#12      ; to 16, the immediate value
;...              ; of the mov instruction is
;                ; interpreted as 0x12.

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

radix   0x0a        ; Reset the default base to 10.
movs    r0,#12      ; Now, the immediate value of
;...              ; the mov instruction is
;                ; interpreted as 0x0c.

end

```

### Controlling case sensitivity

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

```

module caseSensitivity1
section MYCODE:CODE(2)

caseoff
label nop           ; Stored as "LABEL".
b LABEL
end

```

The following will generate a duplicate label error:

```

module caseSensitivity2
section MYCODE:CODE(2)

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

```

---

## Call frame information directives

When you debug an application using C-SPY, you can view the *call stack*, that is, the chain of functions that called the current function. To make this possible when compiling C source code, the compiler supplies debug information that describes the layout of the call frame, in particular information about where the return address is stored.

If you want the call stack to be available when you debug a routine written in assembler language, you must supply equivalent debug information in your assembler source code using the assembler directive `CFI`.

This directive allows backtrace information to be defined in the assembler source 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 a data block to be a conditional thread.
CFI DATAALIGN	Declares data alignment.
CFI DEFAULT	Declares the default state of all resources.
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 FUNCALL	Declares function calls for stack usage analysis.
CFI FUNCTION	Declares a function associated with a data block.
CFI INDIRECTCALL	Declares indirect calls for stack usage analysis.
CFI INVALID	Starts a range of invalid backtrace information.
CFI NAMES	Starts a names block.
CFI NOCALLS	Declares absence of calls for stack usage analysis.
CFI NOFUNCTION	Declares a data block to not be associated with a function.
CFI PICKER	Declares a data block to be a picker thread. Used by the compiler for keeping track of execution paths when code is shared within or between functions.
CFI REMEMBERSTATE	Remembers the backtrace information state.
CFI RESOURCE	Declares a resource.
CFI RESTORESTATE	Restores the saved backtrace information state.
CFI RETURNADDRESS	Declares a return address column.
CFI STACKFRAME	Declares a stack frame CFA.
CFI VALID	Ends a range of invalid backtrace information.
CFI <i>cfa</i>	Declares the value of a CFA.
CFI <i>resource</i>	Declares the value of a resource.

Table 25: Call frame information directives

## SYNTAX

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

### Names block directives

```
CFI NAMES name
CFI ENDNAMES name
CFI RESOURCE resource : bits [, resource : bits] ...
CFI STACKFRAME cfa resource type [, cfa resource type] ...
CFI BASEADDRESS cfa type [, cfa type] ...
```

### Common block directives

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

### 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*

### Stack usage analysis directives

*CFI FUNCALL { caller } callee*

*CFI INDIRECTCALL { caller }*

*CFI NOCALLS { caller }*

### PARAMETERS

<i>bits</i>	The size of the resource in bits.
<i>callee</i>	The label of the called function.
<i>caller</i>	The label of the calling function.
<i>cfa</i>	The name of a CFA (canonical frame address).
<i>cfiexpr</i>	A CFI expression (see <i>Using expressions for complex cases</i> , page 114).
<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 shrinks 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 toward higher addresses, the factor is negative; if it grows toward lower addresses, the factor is positive. 1 is the default, but a larger value shrinks the produced backtrace information in size. The possible ranges are -256 to -1 and 1 to 256.
<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>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 CFI directives provide C-SPY with information about the state of the calling function(s). This *backtrace* information is used for keeping track of the contents of *resources*, such as registers or memory cells, in the assembler code. The most important of this information is the return address, and the value of the stack pointer at the entry of the function or assembler routine.

With this information, C-SPY can reconstruct the state of the calling function, and thereby unwind the stack and show the correct values of registers or other resources before entering the function. This enables 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 application. 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. The stack usage analysis directives are not part of the call frame information. They are just a convenient way for the compiler and the system library to pass call graph information to the linker.

A full description of the calling convention might require extensive call frame information. In many cases, a more limited approach will suffice. When describing the call frame information, the following three components must be present:

- A *names block* describing the available resources to be tracked
- A *common block* corresponding to the calling convention
- A *data block* describing the changes that are performed on the call frame. This typically includes information about when the stack pointer is changed, and when permanent registers are stored or restored on the stack.

The recommended way to create an assembler language routine that handles call frame information correctly is to start with a C skeleton function that you compile to generate assembler output. For an example, see the *IAR C/C++ Development Guide for ARM*.

### Backtrace rows and columns

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

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

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

### Defining a names block

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

Start and end a names block with the directives:

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

where *name* is the name of the block.

Only one names block can be open at a time.

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

- To declare a resource, use this directive:

```
CFI RESOURCE resource : bits
```

The parameters are the name of the resource and the size of the resource in bits. The name must be one of the register names defined in the AEABI document *DWARF for the ARM architecture*.

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

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

```
CFI STACKFRAME cfa resource type
```

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

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

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

```
CFI BASEADDRESS cfa type
```

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

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

## Defining a common block

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

Start a common block with the directive:

```
CFI COMMON name USING namesblock
```

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

Declare the return address column with the directive:

```
CFI RETURNADDRESS resource type
```

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

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

Inside a common block, you can declare the initial value of a CFA or a resource by using the directives listed last in *Common block directives*, page 107. For more information about these directives, see *Rules for simple cases*, page 112 and *Using expressions for complex cases*, page 114.

## Defining a data block

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

Start a data block with the directive:

```
CFI BLOCK name USING commonblock
```

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

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

```
CFI FUNCTION label
```

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

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

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

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

Inside a data block, you can manipulate the values of the columns by using the directives listed last in *Data block directives*, page 107. For more information on these directives, see *Rules for simple cases*, page 112, and *Using expressions for complex cases*, page 114.

## RULES FOR SIMPLE CASES

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

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

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

In those rare cases where the descriptive power of the simple rules are not enough, you can use a full CFI expression to describe the information (see *Using expressions for complex cases*, page 114). 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 107.

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
```

### USING EXPRESSIONS FOR COMPLEX CASES

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

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

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

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

## Unary operators

Overall syntax: *OPERATOR(operand)*

Operator	Operand	Description
COMPLEMENT	<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 26: 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 <i>RSHIFTL</i> , the sign bit is preserved when shifting.

Table 27: 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 27: 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 the memory type <i>type</i> of size <i>size</i> .

Table 28: Ternary operators in CFI expressions

## STACK USAGE ANALYSIS DIRECTIVES

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

The `CFI_FUNCALL` and the `CFI_INDIRECTCALL` directives must be placed where the stack usage information is correct. The easiest way to do this is usually to place them immediately before the instruction that performs the call. The `CFI_NOCALLS` directive can be placed anywhere in the data block.

### EXAMPLE

The following is an example specific to the ARM core. More examples can be obtained by generating assembler output when you compile a C source file.

Consider a Cortex-M3 device with its stack pointer `R13`, link register `R14` and general purpose registers `R0–R12`. Register `R0`, `R2`, `R3` and `R12` will be used as scratch registers (these registers may be destroyed by a function call), whereas register `R1` must be restored after the function call.

Consider the following short code sample with the corresponding backtrace rows and columns. At entry, assume that the register `R14` contains a 32-bit return address. The stack grows from high addresses toward zero. The CFA denotes the top of the call frame, that is, the value of the stack pointer after returning from the function.

Address	CFA	R0	R1	R2	R3	R4-R11	R12	R13	R14	Assembler code
00000000	R13 + 0	—	SAME	—	—	SAME	—	—	SAME	PUSH {r1,lr}
00000002	R13 + 8		CFA - 8						CFA - 4	MOVS r1,#4
00000004										BL func2
00000008										POP {r0,lr}
0000000C	R13 + 0		R0						SAME	MOV r1,r0
0000000E			SAME							BX lr

Table 29: 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 `R13 + 0`. The backtrace row at address `0000` is the initial row and the result of the calling convention used for the function.

The R13 column is empty since the CFA is defined in terms of the stack pointer. The R14 column is the return address column—that is, the location of the return address. The R0 column has a ‘—’ in the first row 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 ArmCore
cfi      stackframe cfa r13 DATA
cfi      resource r0:32, r1:32, r2:32, r3:32
cfi      resource r4:32, r5:32, r6:32, r7:32
cfi      resource r8:32, r9:32, r10:32, r11:32
cfi      resource r12:32, r13:32, r14:32
cfi      endnames ArmCore

```

### Defining the common block

```

cfi      common trivialCommon using ArmCore
cfi      codealign 2
cfi      dataalign 4
cfi      returnaddress r14 CODE
cfi      cfa      r13+0
cfi      default samevalue
cfi      r0      undefined
cfi      r2      undefined
cfi      r3      undefined
cfi      r12     undefined
cfi      endcommon trivialCommon

```

**Note:** `r13` cannot be changed using a CFI directive since it is the resource associated with CFA.

### Defining the data block

```

section MYCODE:CODE(2)

cfi      block trivialBlock using trivialCommon
cfi      function func1

thumb

func1    push    {r1,lr}

cfi      r1     frame(cfa, -8)

```

```
cfi    r14 frame(cfa, -4)
cfi    cfa r13+8

movs   r1,#4

cfi    funcall func2

bl     func2
pop    {r0,lr}

cfi    r1  r0
cfi    r14 samevalue
cfi    cfa r13

mov    r1,r0

cfi    r1 samevalue

bx     lr

cfi    endblock trivialBlock

end
```

**Note:** You should place the CFI directives at the point where the backtrace information has changed, in other words, immediately *after* the instruction that changes the backtrace information.





# Assembler pseudo-instructions

The IAR Assembler for ARM 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 30: 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, MOV <sub>T</sub>	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 30: Pseudo-instructions (Continued)

\* **Deprecated. Use THUMB instead.**

## Descriptions of pseudo-instructions

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

### ADR (ARM)

#### Syntax

`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:

```

name      armAdr
section   MYCODE:CODE(2)
arm
adr       r0,thumbLabel    ; Becomes "add r0,pc,#1".
bx        r0
    
```

```

                                thumb
thumbLabel ; ...

                                end

```

## ADR (CODE16)

Syntax	<code>ADR register, expression</code>
Parameters	<p><i>register</i>      The register to load.</p> <p><i>expression</i>    A program-relative expression that evaluates to a word-aligned address within the range +4 to +1024 bytes.</p>
Description	This Thumb-1 ADR can generate word-aligned addresses only (that is, addresses divisible by 4). Use the <code>ALIGNROM</code> directive to ensure that the address is aligned (unless <code>DC32</code> is used, because it is always word-aligned).

## ADR (THUMB)

Syntax	<code>ADR{condition} register, expression</code>
Parameters	<p><i>{condition}</i>    An optional condition code if the instruction is placed after an <code>IT</code> instruction.</p> <p><i>register</i>        The register to load.</p> <p><i>expression</i>     A program-relative expression that evaluates to an address within the range -4095 to 4095 bytes.</p>
Description	<p>Similar to <code>ADR (CODE16)</code>, but the address range can be larger if a 32-bit Thumb-2 instruction is available in the architecture used.</p> <p>If the address offset is positive and the address is word-aligned, the 16-bit <code>ADR (CODE16)</code> version will be generated by default.</p> <p>The 16-bit version can be specified explicitly with the <code>ADR.N</code> instruction. The 32-bit version can be specified explicitly with the <code>ADR.W</code> instruction.</p>

Example	<pre> name      thumbAdr section  MYCODE:CODE(2) thumb </pre>
---------	---

```

        adr    r0,dataLabel    ; Becomes "add r0,pc,#4".
        add   r0,r0,r1
        bx    lr

        data
        alignrom 2
dataLabel dc32    0xABCD19

        end
    
```

See also

*ADR (CODE16)*, page 123 if only 16-bit Thumb instructions are available.

## ADRL (ARM)

Syntax

*ADRL*{*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 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.

Example

```

        name    armAdrL
        section MYCODE:CODE(2)
        arm
        adrl   r1,label+0x2345 ; Becomes "add r1,pc,#0x45"
                                   ;       and "add r1,r1,#0x2300"

        data
label   dc32    0

        end
    
```

## ADRL (THUMB)

Syntax	<code>ADRL{condition} register,expression</code>
Parameters	<p><code>{condition}</code> An optional condition code if the instruction is placed after an IT instruction.</p> <p><code>register</code> The register to load.</p> <p><code>expression</code> A program-relative expression that evaluates to an address within the range <math>\pm 1</math> Mbyte.</p>
Description	Similar to <code>ADRL</code> (ARM), but the address range can be larger. This instruction is only available in a core supporting the Thumb-2 instruction set.

## LDR (ARM)

Syntax	<code>LDR{condition} register,=expression1</code> or <code>LDR{condition} register,expression2</code>
Parameters	<p><code>condition</code> An optional condition code.</p> <p><code>register</code> The register to load.</p> <p><code>expression1</code> Any 32-bit expression.</p> <p><code>expression2</code> A program location counter-relative expression in the range -4087 to +4103 from the program location counter.</p>
Description	<p>The first form of the <code>LDR</code> 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.</p> <p>If the value of <code>expression1</code> is within the range of a <code>MOV</code> or <code>MVN</code> instruction, the assembler generates the appropriate instruction. If the value of <code>expression1</code> is not within the range of a <code>MOV</code> or <code>MVN</code> instruction, or if the <code>expression1</code> is unsolved, the assembler places the constant in a literal pool and generates a program-relative <code>LDR</code> 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.</p>

Example	<pre> name      armLdr section  MYCODE:CODE(2) arm ldr      r1,=0x12345678 ; Becomes "ldr r1,[pc,#4]":                                 ; loads 0x12345678 from the                                 ; literal pool.                                 ; ldr      r2,label       ; Becomes "ldr r2,[pc,#-4]":                                 ; loads 0xFFEEDDCC into r2.                                 ; data label    dc32    0xFFEEDDCC ltorg                                ; The literal pool is placed                                 ; here. end </pre>
---------	---

See also                   The `LTORG` directive in the section *Assembler control directives*, page 102.

## LDR (CODE16)

Syntax	<pre> LDR <i>register</i>,=<i>expression1</i> or LDR <i>register</i>, <i>expression2</i> </pre>
--------	---

Parameters	<p><i>register</i>           The register to load. LDR can access the low registers (R0–R7) only.</p> <p><i>expression1</i>       Any 32-bit expression.</p> <p><i>expression2</i>       A program location counter-relative expression +4 to +1024 from the program location counter.</p>
------------	--

Description	<p>As in ARM mode, the first form of the <code>LDR</code> 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.</p>
-------------	--

## LDR (THUMB)

Syntax	<pre> LDR{<i>condition</i>} <i>register</i>,=<i>expression</i> </pre>
--------	---

Parameters	<p><i>condition</i>           An optional condition code if the instruction is placed after an <code>IT</code> instruction.</p>
------------	---

*register*            The register to load.  
*expression*        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.

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.

**Example**

```

name      thumbLdr
extern   extLabel
section  MYCODE:CODE(2)
thumb
ldr      r1,=extLabel    ; Becomes "ldr r1,[pc,#8]":
nop                                           ; loads extLabel from the
                                           ; literal pool.
ldr      r2,label       ; Becomes "ldr r2,[pc,#0]":
nop                                           ; loads 0xFFEEDDCC into r2.
data
label    dc32           0xFFEEDDCC
ltorg                                         ; The literal pool is placed
                                           ; here.
end

```

**See also**

LDR (CODE16), page 126 if only 16-bit Thumb instructions are available.

## MOV (CODE16)

Syntax	<code>MOV Rd, Rs</code>
Parameters	<p><code>Rd</code>            The destination register.</p> <p><code>Rs</code>            The source register.</p>
Description	<p>The Thumb <code>MOV</code> pseudo-instruction moves the value of a low register to another low register (<code>R0-R7</code>). The Thumb <code>MOV</code> instruction cannot move values from one low register to another.</p> <p><b>Note:</b> The <code>ADD</code> immediate instruction generated by the assembler has the side-effect of updating the condition codes.</p> <p>The <code>MOV</code> pseudo-instruction uses an <code>ADD</code> immediate instruction with a zero immediate value.</p> <p><b>Note:</b> This description is only valid when using the <code>CODE16</code> directive. After the <code>THUMB</code> directive, the interpretation of the instruction syntax is defined by the Unified Assembler syntax from Advanced RISC Machines Ltd.</p>
Example	<code>MOV r2,r3 ; generates the opcode for ADD r2,r3,#0</code>

## MOV32 (THUMB)

Syntax	<code>MOV32{condition} register, expression</code>
Parameters	<p><i>condition</i>        An optional condition code if the instruction is placed after an <code>IT</code> instruction.</p> <p><i>register</i>            The register to load.</p> <p><i>expression</i>        Any 32-bit expression.</p>
Description	<p>Similar to the <code>LDR (THUMB)</code> instruction, but will load the constant by generating a pair of the <code>MOV (MOVW)</code> and the <code>MOVT</code> instructions.</p> <p>This pseudo-instruction always generates two 32-bit instructions and it is only available in a core supporting the Thumb-2 instruction set.</p>



## NOP (ARM)

Syntax

`NOP`

Description

`NOP` generates the preferred ARM no-operation code:

```
MOV r0, r0
```

**Note:** `NOP` is not a pseudo-instruction in architecture versions that include a `NOP` instruction (ARMv6K, ARMv6T2, ARMv7).

## NOP (CODE16)

Syntax

`NOP`

Description

`NOP` generates the preferred Thumb no-operation code:

```
MOV r8, r8
```

**Note:** `NOP` is not a pseudo-instruction in architecture versions that include a `NOP` instruction (ARMv6T2, ARMv7).



# 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, and 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 are 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 IAR Assembler for ARM 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 47
- Set the number of maximum errors before the compilation stops, see *-E*, page 37.

### ASSEMBLY WARNING MESSAGES

Assembly warning messages are produced when the assembler finds 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 finds a construct which violates the language rules.

## **ASSEMBLY FATAL ERROR MESSAGES**

Assembly fatal error messages are produced when the assembler finds a user error so severe that further processing is not considered meaningful. After the diagnostic message is issued, the assembly is immediately ended. 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 was a serious and unexpected failure due to a fault in the assembler.

During assembly, several internal consistency checks are performed and if any of these checks fail, the assembler terminates 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 Systems Technical Support. Please include information enough to reproduce the problem. This would typically include:

- The product name
- The version number of the assembler, which can be seen in the header of the list files generated by the assembler
- Your license number
- The exact internal error message text
- The source file of the program that generated the internal error
- A list of the options that were used when the internal error occurred.

# Migrating to the IAR Assembler for ARM

Assembly source code that was originally written for assemblers from other vendors can also be used with the IAR Assembler for ARM. 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 IAR Assembler for ARM.

---

## Introduction

The IAR Assembler for ARM (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 IAR Assembler for ARM 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)
arm
```

The two instructions below are interpreted differently by ARMASM and IASMARM. ICCARM interprets a reference to `T` as an odd address (with the Thumb mode bit set), but in ARMASM it is even (the Thumb mode bit is not set).

```
adr    r0, T+1
mov    r1, #T-.
```

To achieve the same interpretation for both ARMASM and ICCARM, use `:OR:` to set the Thumb mode bit, or `:AND:` to clear it:

```
add    r0, pc, # (T-.-8) :OR: 1
mov    r1, # (T-.) :AND: ~1

thumb
T      nop
end
```

---

## Alternative register names

The IAR Assembler for ARM 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 assembler register names:

Alternative register name	Assembler register name
A1	R0
A2	R1
A3	R2
A4	R3
V1	R4
V2	R5
V3	R6
V4	R7
V5	R8
V6	R9
V7	R10
SB	R9

*Table 31: Alternative register names*

Alternative register name	Assembler register name
SL	R10
FP	R11
IP	R12

Table 31: Alternative register names (Continued)

For further descriptions of the registers, see *Register symbols*, page 25.

## Alternative mnemonics

A number of mnemonics used by other assemblers will be translated by the 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	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
BNNE	BEQ
BNPL	BMI
BNVC	BVS

Table 32: Alternative mnemonics

Alternative mnemonic	Assembler mnemonic
BNVS	BVC
CMN{cond}S	CMN{cond}
CMP{cond}S	CMP{cond}
EORS	EOR
LSL	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 32: 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 assembler when the option `-j` is specified. The following operator synonyms are allowed in both ARM and Thumb modes:

Operator synonym	Assembler operator
:AND:	&
:EOR:	^
:LAND:	&&
:LEOR:	XOR
:LNOT:	!
:LOR:	
:MOD:	%

Table 33: Operator synonyms



Operator synonym	Assembler operator
:NOT:	~
:OR:	
:SHL:	<<
:SHR:	>>

Table 33: Operator synonyms (Continued)

**Note:** In some cases, assembler operators and operator synonyms have different precedence levels. For further descriptions of the operators, see the chapter *Assembler operators*, page 49.

## Warning messages

Unless the option `-j` is specified, the 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) . . . . . 67
- absolute expressions . . . . . 28
- ADD (assembler instruction) . . . . . 122
- ADD (CFI operator) . . . . . 115
- addition (assembler operator) . . . . . 53
- address field, in assembler list file . . . . . 29
- addresses, loading into a register . . . . . 122–125
- ADR (ARM) (pseudo-instruction) . . . . . 122
- ADR (CODE16) (pseudo-instruction) . . . . . 123
- ADR (THUMB) (pseudo-instruction) . . . . . 123
- ADRL (ARM) (pseudo-instruction) . . . . . 124
- ADRL (THUMB) (pseudo-instruction) . . . . . 125
- ALIAS (assembler directive) . . . . . 78
- alignment, of sections . . . . . 76
- ALIGNRAM (assembler directive) . . . . . 74
- ALIGNROM (assembler directive) . . . . . 74
- :AND: (assembler operator) . . . . . 55
- AND (CFI operator) . . . . . 115
- architecture, ARM . . . . . 11
- \_args (assembler directive) . . . . . 82
- \_args (predefined macro symbol) . . . . . 85
- ARM architecture and instruction set . . . . . 11
- ARM (assembler directive) . . . . . 72
- ARMASM assembler . . . . . 133
- \_\_ARMVFP\_\_ (predefined symbol) . . . . . 26
- \_\_ARM\_ADVANCED\_SIMD\_\_ (predefined symbol) . . . . . 26
- \_\_ARM\_MEDIA\_\_ (predefined symbol) . . . . . 26
- \_\_ARM\_MPCORE\_\_ (predefined symbol) . . . . . 26
- \_\_ARM\_PROFILE\_M\_\_ (predefined symbol) . . . . . 26
- ASCII character constants . . . . . 23
- asm (filename extension) . . . . . 19
- assembler control directives . . . . . 102
- assembler diagnostics . . . . . 131
- assembler directives
  - assembler control . . . . . 102
  - call frame information (CFI) . . . . . 105
  - conditional assembly . . . . . 80
    - See also* C-style preprocessor directives
  - C-style preprocessor . . . . . 94
  - data definition or allocation . . . . . 99
  - list file control . . . . . 90
  - macro processing . . . . . 82
  - module control . . . . . 67
  - segment control . . . . . 74
  - summary . . . . . 63
  - symbol control . . . . . 70
  - value assignment . . . . . 78
- assembler environment variables . . . . . 20
- assembler expressions . . . . . 22
- assembler instructions . . . . . 21
  - ADD . . . . . 122
  - BX . . . . . 73
  - LDR . . . . . 125
  - MOV . . . . . 125
  - MVN . . . . . 125
  - SUB . . . . . 122
- assembler invocation syntax . . . . . 19
- assembler labels . . . . . 24
  - format of . . . . . 21
  - in Thumb code . . . . . 133
- assembler list files
  - address field . . . . . 29
  - comments . . . . . 103
  - conditional code and strings . . . . . 91
  - cross-references
    - generating (LSTXRF) . . . . . 92
    - generating (-x) . . . . . 48
  - data field . . . . . 29
  - enabling and disabling (LSTOUT) . . . . . 91
  - filename, specifying (-l) . . . . . 41
  - generated lines, controlling (LSTREP) . . . . . 91
  - generating (-L) . . . . . 41
  - header section, omitting (-N) . . . . . 43
  - #include files, specifying (-i) . . . . . 40
  - lines per page, specifying (-p) . . . . . 45

macro execution information, including (-B) . . . . .	35
macro-generated lines, controlling . . . . .	91
symbol and cross-reference table . . . . .	30
tab spacing, specifying . . . . .	46
using directives to format . . . . .	92
assembler macros	
arguments, passing to . . . . .	85
defining . . . . .	83
generated lines, controlling in list file . . . . .	91
in-line routines . . . . .	87
predefined symbol . . . . .	85
processing . . . . .	86
quote characters, specifying . . . . .	42
special characters, using . . . . .	84
assembler object file, specifying filename . . . . .	44
assembler operators . . . . .	49
in expressions . . . . .	22
precedence . . . . .	49
assembler options	
passing to assembler . . . . .	20
command line, setting . . . . .	33
extended command file, setting . . . . .	33
summary . . . . .	34
assembler output, including debug information . . . . .	45
assembler pseudo-instructions . . . . .	121
assembler source code, porting . . . . .	73
assembler source files, including . . . . .	97, 104
assembler source format . . . . .	21
assembler symbols . . . . .	24
exporting . . . . .	71
importing . . . . .	71
in relocatable expressions . . . . .	28
predefined . . . . .	25
undefining . . . . .	47
redefining . . . . .	79
assembling, invocation syntax . . . . .	19
assembly error messages . . . . .	132
assembly messages format . . . . .	131

assembly warning messages . . . . .	131
disabling . . . . .	47
ASSIGN (assembler directive) . . . . .	78
assumptions (programming experience) . . . . .	11

## B

-B (assembler option) . . . . .	35
backtrace information <i>See</i> call frame information	
backtrace information, defining . . . . .	105
bitwise AND (assembler operator) . . . . .	55
bitwise exclusive OR (assembler operator) . . . . .	56
bitwise NOT (assembler operator) . . . . .	56
bitwise OR (assembler operator) . . . . .	56
bold style, in this guide . . . . .	14
__BUILD_NUMBER__ (predefined symbol) . . . . .	26
BX (assembler instruction) . . . . .	73
byte order . . . . .	27
BYTE1 (assembler operator) . . . . .	58
BYTE2 (assembler operator) . . . . .	58
BYTE3 (assembler operator) . . . . .	58
BYTE4 (assembler operator) . . . . .	58

## C

-c (assembler option) . . . . .	36
call frame information . . . . .	105
call frame information directives . . . . .	105
call stack . . . . .	105
case sensitive user symbols . . . . .	46
case sensitivity, controlling . . . . .	103
CASEOFF (assembler directive) . . . . .	102
CASEON (assembler directive) . . . . .	102
CFI BASEADDRESS (assembler directive) . . . . .	106
CFI BLOCK (assembler directive) . . . . .	106
CFI cfa (assembler directive) . . . . .	106
CFI CODEALIGN (assembler directive) . . . . .	106
CFI COMMON (assembler directive) . . . . .	106
CFI CONDITIONAL (assembler directive) . . . . .	106

- CFI DATAALIGN (assembler directive) . . . . . 106
  - CFI DEFAULT (assembler directive) . . . . . 106
  - CFI directives . . . . . 105
  - CFI ENDBLOCK (assembler directive) . . . . . 106
  - CFI ENDCOMMON (assembler directive) . . . . . 106
  - CFI ENDNAMES (assembler directive) . . . . . 106
  - CFI expressions . . . . . 114
  - CFI FRAMECELL (assembler directive) . . . . . 106
  - CFI FUNCALL (assembler directive) . . . . . 106
  - CFI FUNCTION (assembler directive) . . . . . 106
  - CFI INDIRECTCALL (assembler directive) . . . . . 106
  - CFI INVALID (assembler directive) . . . . . 106
  - CFI NAMES (assembler directive) . . . . . 106
  - CFI NOCALLS (assembler directive) . . . . . 106
  - CFI NOFUNCTION (assembler directive) . . . . . 106
  - CFI operators . . . . . 115
  - CFI PICKER (assembler directive) . . . . . 106
  - CFI REMEMBERSTATE (assembler directive) . . . . . 106
  - CFI RESOURCE (assembler directive) . . . . . 106
  - CFI resource (assembler directive) . . . . . 106
  - CFI RESTORESTATE (assembler directive) . . . . . 106
  - CFI RETURNADDRESS (assembler directive) . . . . . 106
  - CFI STACKFRAME (assembler directive) . . . . . 106
  - CFI VALID (assembler directive) . . . . . 106
  - CFI (assembler directive) . . . . . 105
  - character constants, ASCII . . . . . 23
  - CLIB, documentation . . . . . 13
  - CODE16 (assembler directive) . . . . . 72
  - CODE32 (assembler directive) . . . . . 72
  - COL (assembler directive) . . . . . 90
  - command line error messages, assembler . . . . . 131
  - command line options . . . . . 33
    - part of invocation syntax . . . . . 19
    - passing . . . . . 20
    - typographic convention . . . . . 14
  - command line, extending . . . . . 38
  - command prompt icon, in this guide . . . . . 14
  - comments
    - in assembler list file . . . . . 103
    - in assembler source code . . . . . 21
    - in C-style preprocessor directives . . . . . 97
    - multi-line, using with assembler directives . . . . . 104
  - common block (call frame information) . . . . . 109
  - COMPLEMENT (CFI operator) . . . . . 115
  - computer style, typographic convention . . . . . 14
  - conditional assembly directives . . . . . 80
    - See also* C-style preprocessor directives
  - conditional code and strings, listing . . . . . 91
  - constants
    - default base of . . . . . 103
    - integer . . . . . 22
  - conventions, used in this guide . . . . . 14
  - copyright notice . . . . . 2
  - cpu (assembler option) . . . . . 36
  - CPU, defining in assembler. *See* processor configuration
  - CRC, in assembler list file . . . . . 30
  - cross-references, in assembler list file
    - generating (LSTXRF) . . . . . 92
    - generating (-x) . . . . . 48
  - current time/date (assembler operator) . . . . . 58
  - C-style preprocessor directives . . . . . 94
  - C++ terminology . . . . . 14
- ## D
- D (assembler option) . . . . . 37
  - data allocation directives . . . . . 99
  - data block (call frame information) . . . . . 109
  - data definition directives . . . . . 99
  - data field, in assembler list file . . . . . 29
  - DATA (assembler directive) . . . . . 72
  - data, defining in Thumb code section . . . . . 73
  - \_\_DATE\_\_ (predefined symbol) . . . . . 26
  - DATE (assembler operator) . . . . . 58
  - DCB (assembler directive) . . . . . 99
  - DCD (assembler directive) . . . . . 100
  - DCW (assembler directive) . . . . . 100
  - DC8 (assembler directive) . . . . . 99

DC16 (assembler directive) . . . . .	100
DC24 (assembler directive) . . . . .	100
DC32 (assembler directive) . . . . .	100
debug information, including in assembler output . . . . .	45
default base, for constants . . . . .	103
#define (assembler directive) . . . . .	94
DEFINE (assembler directive) . . . . .	78
DF32 (assembler directive) . . . . .	100
DF64 (assembler directive) . . . . .	100
diagnostic messages, options for . . . . .	131
diagnostics . . . . .	131
directives. <i>See</i> assembler directives . . . . .	2
disclaimer . . . . .	2
DIV (CFI operator) . . . . .	115
division (assembler operator) . . . . .	53
DLIB, documentation . . . . .	13
document conventions . . . . .	14
documentation, overview of guides . . . . .	12
DS (assembler directive) . . . . .	100
DS8 (assembler directive) . . . . .	100
DS16 (assembler directive) . . . . .	100
DS24 (assembler directive) . . . . .	100
DS32 (assembler directive) . . . . .	100

## E

-E (assembler option) . . . . .	37
-e (assembler option) . . . . .	38
edition, of this guide . . . . .	2
efficient coding techniques . . . . .	30
#elif (assembler directive) . . . . .	94
#else (assembler directive) . . . . .	94
ELSE (assembler directive) . . . . .	80
ELSEIF (assembler directive) . . . . .	80
END (assembler directive) . . . . .	67
--endian (assembler option) . . . . .	38
#endif (assembler directive) . . . . .	94
ENDIF (assembler directive) . . . . .	80
ENDM (assembler directive) . . . . .	82

ENDR (assembler directive) . . . . .	82
environment variables . . . . .	
assembler . . . . .	20
IASMARM . . . . .	20
IASMARM_INC . . . . .	20
:EOR: (assembler operator) . . . . .	56
EQ (CFI operator) . . . . .	115
EQU (assembler directive) . . . . .	78
equal (assembler operator) . . . . .	54
#error (assembler directive) . . . . .	95
error messages . . . . .	
format . . . . .	131
maximum number, specifying . . . . .	37
#error, using to display . . . . .	97
EVEN (assembler directive) . . . . .	74
EXITM (assembler directive) . . . . .	82
experience, programming . . . . .	11
expressions . . . . .	22
extended command line file (extend.xcl) . . . . .	33, 38
EXTERN (assembler directive) . . . . .	70
EXTWEAK (assembler directive) . . . . .	70

## F

-f (assembler option) . . . . .	33, 38
false value, in assembler expressions . . . . .	24
fatal errors . . . . .	132
__FILE__ (predefined symbol) . . . . .	26
file extensions. <i>See</i> filename extensions . . . . .	
file types . . . . .	
assembler output . . . . .	19
assembler source . . . . .	19
extended command line . . . . .	33, 38
#include, specifying path . . . . .	40
filename extensions . . . . .	
asm . . . . .	19
msa . . . . .	19
o . . . . .	19
s . . . . .	19

xcl ..... 33, 38  
 filenames, specifying for assembler object file ..... 44  
 first byte (assembler operator) ..... 58  
 floating-point constants ..... 23  
 floating-point coprocessor, defining in assembler ..... 39  
 formats  
   assembler source code ..... 21  
   diagnostic messages ..... 131  
   in list files ..... 29  
 fourth byte (assembler operator) ..... 58  
 --fpu (assembler option) ..... 39  
 FRAME (CFI operator) ..... 116

## G

-G (assembler option) ..... 39  
 -g (assembler option) ..... 39  
 GE (CFI operator) ..... 115  
 global value, defining ..... 79  
 greater than or equal (assembler operator) ..... 55  
 greater than (assembler operator) ..... 55  
 GT (CFI operator) ..... 115

## H

header files, SFR. .... 30  
 header section, omitting from assembler list file ..... 43  
 high byte (assembler operator) ..... 59  
 high word (assembler operator) ..... 59  
 HIGH (assembler operator) ..... 59  
 HWRD (assembler operator) ..... 59

## I

-I (assembler option) ..... 40  
 \_\_IAR\_SYSTEMS\_ASM\_\_ (predefined symbol) ..... 26  
 \_\_IASMARM\_\_ (predefined symbol) ..... 26  
 IASMARM (environment variable) ..... 20  
 IASMARM\_INC (environment variable) ..... 20

icons, in this guide ..... 14  
 #if (assembler directive) ..... 95  
 IF (assembler directive) ..... 80  
 IF (CFI operator) ..... 116  
 #ifdef (assembler directive) ..... 95  
 #ifndef (assembler directive) ..... 95  
 IMPORT (assembler directive) ..... 70  
 #include files ..... 40  
 #include files, specifying ..... 40  
 #include (assembler directive) ..... 95  
 include files, disabling search for ..... 39  
 include paths, specifying ..... 40  
 INCLUDE (assembler directive) ..... 102  
 installation directory ..... 14  
 instruction set, ARM ..... 11  
 integer constants ..... 22  
 internal errors, assembler ..... 132  
 invocation syntax ..... 19  
 in-line coding, using macros ..... 87  
 italic style, in this guide ..... 14

## J

-j (assembler option) ..... 41

## L

-L (assembler option) ..... 41  
 -l (assembler option) ..... 41  
 labels. *See* assembler labels  
 :LAND: (assembler operator) ..... 55  
 LDR (ARM) (pseudo-instruction) ..... 125  
 LDR (assembler instruction) ..... 125  
 LDR (CODE16) (pseudo-instruction) ..... 126  
 LDR (THUMB) (pseudo-instruction) ..... 126  
 LE (CFI operator) ..... 115  
 --legacy (assembler option) ..... 42  
 :LEOR: (assembler operator) ..... 62  
 less than or equal (assembler operator) ..... 54

less than (assembler operator) . . . . .	54
LIBRARY (assembler directive) . . . . .	66
lightbulb icon, in this guide . . . . .	14
__LINE__ (predefined symbol) . . . . .	26
#line (assembler directive) . . . . .	95
lines per page, in assembler list file . . . . .	45
list file format . . . . .	29
body . . . . .	29
CRC . . . . .	30
header . . . . .	29
symbol and cross reference . . . . .	
list files . . . . .	
control directives for . . . . .	90
controlling contents of (-c) . . . . .	36
cross-references, generating (-x) . . . . .	48
filename, specifying (-I) . . . . .	41
generating (-L) . . . . .	41
header section, omitting (-N) . . . . .	43
#include files, specifying (-i) . . . . .	40
literal pool . . . . .	125
LITERAL (CFI operator) . . . . .	115
__LITTLE_ENDIAN__ (predefined symbol) . . . . .	27
:LNOT: (assembler operator) . . . . .	57
LOAD (CFI operator) . . . . .	116
local value, defining . . . . .	78
LOCAL (assembler directive) . . . . .	82
logical AND (assembler operator) . . . . .	55
logical exclusive OR (assembler operator) . . . . .	62
logical NOT (assembler operator) . . . . .	57
logical OR (assembler operator) . . . . .	57
logical shift left (assembler operator) . . . . .	57
logical shift right (assembler operator) . . . . .	57
:LOR: (assembler operator) . . . . .	57
low byte (assembler operator) . . . . .	59
low register values, moving . . . . .	128
low word (assembler operator) . . . . .	59
LOW (assembler operator) . . . . .	59
LSHIFT (CFI operator) . . . . .	115
LSTCND (assembler directive) . . . . .	90

LSTCOD (assembler directive) . . . . .	90
LSTEXP (assembler directives) . . . . .	90
LSTMAC (assembler directive) . . . . .	90
LSTOUT (assembler directive) . . . . .	90
LSTPAG (assembler directive) . . . . .	90
LSTREP (assembler directive) . . . . .	90
LSTXRF (assembler directive) . . . . .	90
LT (CFI operator) . . . . .	115
LTORG (assembler directive) . . . . .	102
LWRD (assembler operator) . . . . .	59

## M

-M (assembler option) . . . . .	42
macro execution information, including in list file . . . . .	35
macro processing directives . . . . .	82
macro quote characters . . . . .	84
specifying . . . . .	42
MACRO (assembler directive) . . . . .	82
macros. <i>See</i> assembler macros . . . . .	
memory space, reserving and initializing . . . . .	101
memory, reserving space in . . . . .	99
#message (assembler directive) . . . . .	95
messages, excluding from standard output stream . . . . .	45
migration to the ARM IAR Assembler . . . . .	133
alternative mnemonics . . . . .	135
alternative register names . . . . .	134
operator synonyms . . . . .	136
warning messages . . . . .	137
migration, from earlier IAR compilers . . . . .	13
MISRA C, documentation . . . . .	13
:MOD: (assembler operator) . . . . .	56
MOD (CFI operator) . . . . .	115
mode control directives . . . . .	72
module consistency . . . . .	69
module control directives . . . . .	67
modules, beginning . . . . .	68
MOV (assembler instruction) . . . . .	125
MOV (CODE16) (pseudo-instruction) . . . . .	128



MOV (THUMB) (pseudo-instruction) . . . . .	128
msa (filename extension) . . . . .	19
MUL (CFI operator) . . . . .	115
multibyte character support . . . . .	43
multiplication (assembler operator) . . . . .	52
MVN (assembler instruction) . . . . .	125

## N

-N (assembler option) . . . . .	43
-n (assembler option) . . . . .	43
NAME (assembler directive) . . . . .	68
names block (call frame information) . . . . .	109
naming conventions . . . . .	15
NE (CFI operator) . . . . .	115
NOP (ARM) (pseudo-instruction) . . . . .	129
NOP (CODE16) (pseudo-instruction) . . . . .	129
:NOT: (assembler operator) . . . . .	56
not equal (assembler operator) . . . . .	54
NOT (CFI operator) . . . . .	115
no-operation code, generating . . . . .	129

## O

-O (assembler option) . . . . .	44
-o (assembler option) . . . . .	44
o (filename extension) . . . . .	19
ODD (assembler directive) . . . . .	74
operands	
format of . . . . .	21
in assembler expressions . . . . .	22
operations, format of . . . . .	21
operation, silent . . . . .	45
operators. <i>See</i> assembler operators	
option summary . . . . .	34
:OR: (assembler operator) . . . . .	56
OR (CFI operator) . . . . .	115

## P

-p (assembler option) . . . . .	45
PAGE (assembler directive) . . . . .	90
PAGSIZ (assembler directive) . . . . .	90
parameters, typographic convention . . . . .	14
part number, of this guide . . . . .	2
porting assembler source code . . . . .	73
#pragma (assembler directive) . . . . .	95
precedence, of assembler operators . . . . .	49
predefined register symbols . . . . .	25
predefined symbols . . . . .	25
in assembler macros . . . . .	85
undefining . . . . .	47
preprocessor symbols	
defining and undefining . . . . .	96
defining on command line . . . . .	37
prerequisites (programming experience) . . . . .	11
PRESERVE8 (assembler directive) . . . . .	68
processor mode, directives . . . . .	72
program location counter (PLC) . . . . .	24
PROGRAM (assembler directive) . . . . .	68
programming experience, required . . . . .	11
programming hints . . . . .	30
pseudo-instructions . . . . .	121
PUBLIC (assembler directive) . . . . .	70
publication date, of this guide . . . . .	2
PUBWEAK (assembler directive) . . . . .	70

## R

-r (assembler option) . . . . .	45
RADIX (assembler directive) . . . . .	102
reference information, typographic convention . . . . .	14
registered trademarks . . . . .	2
registers . . . . .	25
alternative names of . . . . .	134
relocatable expressions . . . . .	28
repeating statements . . . . .	86

REPT (assembler directive) . . . . .	82
REPTC (assembler directive) . . . . .	82
REPTI (assembler directive) . . . . .	82
REQUIRE (assembler directive) . . . . .	70
REQUIRE8 (assembler directive) . . . . .	68
RSEG (assembler directive) . . . . .	74
RSHIFTA (CFI operator) . . . . .	115
RSHIFTL (CFI operator) . . . . .	116
RTMODEL (assembler directive) . . . . .	68
rules, in CFI directives . . . . .	112
runtime model attributes, declaring . . . . .	69

## S

-S (assembler option) . . . . .	45
-s (assembler option) . . . . .	46
s (filename extension) . . . . .	19
second byte (assembler operator) . . . . .	58
SECTION (assembler directive) . . . . .	74
sections	
aligning . . . . .	76
beginning . . . . .	76
SECTION_TYPE (assembler directive) . . . . .	74
segment begin (assembler operator) . . . . .	59
segment control directives . . . . .	74
segment end (assembler operator) . . . . .	60
segment size (assembler operator) . . . . .	61
SET (assembler directive) . . . . .	78
SETA (assembler directive) . . . . .	78
SFB (assembler operator) . . . . .	59
SFE (assembler operator) . . . . .	60
SFR. <i>See</i> special function registers	
:SHL: (assembler operator) . . . . .	57
:SHR: (assembler operator) . . . . .	57
silent operation, specifying in assembler . . . . .	45
simple rules, in CFI directives . . . . .	112
SIZEOF (assembler operator) . . . . .	61
source files	
example of including . . . . .	104

including . . . . .	97
source format, assembler . . . . .	21
source line numbers, changing . . . . .	98
special function registers . . . . .	30
standard input stream (stdin), reading from . . . . .	39
standard output stream, disabling messages to . . . . .	45
statements, repeating . . . . .	86
SUB (assembler instruction) . . . . .	122
SUB (CFI operator) . . . . .	116
subtraction (assembler operator) . . . . .	53
symbol and cross-reference table, in assembler list file . . . . .	30
<i>See also</i> Include cross-reference	
symbol control directives . . . . .	70
symbols	
<i>See also</i> assembler symbols	
exporting to other modules . . . . .	71
predefined, in assembler . . . . .	25
predefined, in assembler macro . . . . .	85
user-defined, case sensitive . . . . .	46
system include files, disabling search for . . . . .	39
--system_include_dir (assembler option) . . . . .	46

## T

-t (assembler option) . . . . .	46
tab spacing, specifying in assembler list file . . . . .	46
target core, specifying. <i>See</i> processor configuration	
temporary values, defining . . . . .	78
terminology . . . . .	14
third byte (assembler operator) . . . . .	58
THUMB (assembler directive) . . . . .	72
__TID__ (predefined symbol) . . . . .	27
__TIME__ (predefined symbol) . . . . .	27
time-critical code . . . . .	87
tools icon, in this guide . . . . .	14
trademarks . . . . .	2
true value, in assembler expressions . . . . .	24
typographic conventions . . . . .	14

## U

-U (assembler option)	47
UGT (assembler operator)	61
ULT (assembler operator)	61
UMINUS (CFI operator)	115
unary minus (assembler operator)	53
unary plus (assembler operator)	53
#undef (assembler directive)	95
unsigned greater than (assembler operator)	61
unsigned less than (assembler operator)	61
user symbols, case sensitive	46

## V

value assignment directives	78
values, defining	99
VAR (assembler directive)	78
__VER__ (predefined symbol)	27
version number, of this guide	2

## W

-w (assembler option)	47
warnings	131
disabling	47
warnings icon, in this guide	14
web sites, recommended	13

## X

-x (assembler option)	48
xcl (filename extension)	33, 38
XOR (assembler operator)	62
XOR (CFI operator)	116

# Symbols

^ (assembler operator)	56
------------------------	----

_args (assembler directive)	82
_args (predefined macro symbol)	85
__ARMVFP__ (predefined symbol)	26
__ARM_ADVANCED_SIMD__ (predefined symbol)	26
__ARM_MEDIA__ (predefined symbol)	26
__ARM_MPCORE__ (predefined symbol)	26
__ARM_PROFILE_M__ (predefined symbol)	26
__BUILD_NUMBER__ (predefined symbol)	26
__DATE__ (predefined symbol)	26
__FILE__ (predefined symbol)	26
__IAR_SYSTEMS_ASM__ (predefined symbol)	26
__IASMARM__ (predefined symbol)	26
__LINE__ (predefined symbol)	26
__LITTLE_ENDIAN__ (predefined symbol)	27
__TID__ (predefined symbol)	27
__TIME__ (predefined symbol)	27
__VER__ (predefined symbol)	27
- (assembler operator)	53
-B (assembler option)	35
-c (assembler option)	36
-D (assembler option)	37
-E (assembler option)	37
-e (assembler option)	38
-f (assembler option)	33, 38
-G (assembler option)	39
-g (assembler option)	39
-I (assembler option)	40
-i (assembler option)	40
-j (assembler option)	41, 133
-L (assembler option)	41
-l (assembler option)	41
-M (assembler option)	42
-N (assembler option)	43
-n (assembler option)	43
-O (assembler option)	44
-o (assembler option)	44
-p (assembler option)	45
-r (assembler option)	45
-S (assembler option)	45

-s (assembler option) . . . . .	46	#include files, specifying . . . . .	40
-t (assembler option) . . . . .	46	#include (assembler directive) . . . . .	95
-U (assembler option) . . . . .	47	#line (assembler directive) . . . . .	95
-w (assembler option) . . . . .	47	#message (assembler directive) . . . . .	95
-x (assembler option) . . . . .	48	#pragma (assembler directive) . . . . .	95
--cpu (assembler option) . . . . .	36	#undef (assembler directive) . . . . .	95
--endian (assembler option) . . . . .	38	+ (assembler operator) . . . . .	53
--fpu (assembler option) . . . . .	39	< (assembler operator) . . . . .	54
--legacy (assembler option) . . . . .	42	<< (assembler operator) . . . . .	57
--system_include_dir (assembler option) . . . . .	46	<= (assembler operator) . . . . .	54
:AND: (assembler operator) . . . . .	55	<> (assembler operator) . . . . .	54
:EOR: (assembler operator) . . . . .	56	= (assembler directive) . . . . .	78
:LAND: (assembler operator) . . . . .	55	= (assembler operator) . . . . .	54
:LEOR: (assembler operator) . . . . .	62	== (assembler operator) . . . . .	54
:LNOT: (assembler operator) . . . . .	57	> (assembler operator) . . . . .	55
:LOR: (assembler operator) . . . . .	57	>= (assembler operator) . . . . .	55
:MOD: (assembler operator) . . . . .	56	>> (assembler operator) . . . . .	57
:NOT: (assembler operator) . . . . .	56	(assembler operator) . . . . .	56
:OR: (assembler operator) . . . . .	56	(assembler operator) . . . . .	57
:SHL: (assembler operator) . . . . .	57	~ (assembler operator) . . . . .	56
:SHR: (assembler operator) . . . . .	57	\$ (assembler directive) . . . . .	102
! (assembler operator) . . . . .	57	\$ (program location counter) . . . . .	24
!= (assembler operator) . . . . .	54		
() (assembler operator) . . . . .	52		
* (assembler operator) . . . . .	52		
/ (assembler operator) . . . . .	53		
/*...*/ (assembler directive) . . . . .	102		
// (assembler directive) . . . . .	102		
& (assembler operator) . . . . .	55		
&& (assembler operator) . . . . .	55		
#define (assembler directive) . . . . .	94		
#elif (assembler directive) . . . . .	94		
#else (assembler directive) . . . . .	94		
#endif (assembler directive) . . . . .	94		
#error (assembler directive) . . . . .	95		
#if (assembler directive) . . . . .	95		
#ifdef (assembler directive) . . . . .	95		
#ifndef (assembler directive) . . . . .	95		
#include files. . . . .	40		

## Numerics

32-bit expressions, loading in register. . . . .	125
--	-----