IAR Embedded Workbench®

IAR Assembler™ Reference Guide

for Advanced RISC Machines Ltd’s
ARM Cores
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
- Contents ........................................................................................................................ 9

## Preface
- Who should read this guide ............................................................... 11
- How to use this guide ........................................................................... 11
- What this guide contains ..................................................................... 12
- Other documentation ........................................................................... 12
  - User and reference guides ................................................................. 12
  - The online help system ................................................................. 13
  - Web sites ............................................................................................ 13
- Document conventions .......................................................................... 14
  - Typographic conventions ................................................................. 14
  - Naming conventions ........................................................................ 15

## Introduction to the IAR Assembler for ARM
- Introduction to assembler programming .............................................. 17
  - Getting started .................................................................................... 18
  - Modular programming ........................................................................ 18
  - External interface details .................................................................... 19
    - Assembler invocation syntax .......................................................... 19
    - Passing options ............................................................................... 20
    - Environment variables ..................................................................... 20
    - Error return codes ........................................................................ 20
- Source format ........................................................................................ 21
- Assembler instructions ........................................................................ 21
  - Expressions, operands, and operators .............................................. 22
    - Integer constants ........................................................................... 22
    - ASCII character constants ............................................................ 23
    - Floating-point constants ................................................................. 23
    - TRUE and FALSE ............................................................................ 24
    - Symbols ........................................................................................... 24
    - Labels ............................................................................................ 24
Assembler directives ................................................................. 63

Summary of assembler directives ........................................... 63

Module control directives .......................................................... 67
Syntax ...................................................................................... 68
Parameters ............................................................................... 68
Descriptions ............................................................................ 68

Symbol control directives .......................................................... 70
Syntax ...................................................................................... 70
Parameters ............................................................................... 71
Descriptions ............................................................................ 71
Examples ................................................................................ 72

Mode control directives ............................................................. 72
Syntax ...................................................................................... 73
Description ............................................................................. 73
Examples ................................................................................ 73

Section control directives .......................................................... 74
Syntax ...................................................................................... 75
Parameters ............................................................................... 75
Descriptions ............................................................................ 76
Examples ................................................................................ 77

Value assignment directives ....................................................... 78
Syntax ...................................................................................... 78
Parameters ............................................................................... 78
Descriptions ............................................................................ 78
Examples ................................................................................ 79

Conditional assembly directives ............................................... 80
Syntax ...................................................................................... 81
Parameters ............................................................................... 81
Descriptions ............................................................................ 81
Examples ................................................................................ 81

Macro processing directives ....................................................... 82
Syntax ...................................................................................... 82
Parameters ............................................................................... 83
Assembler pseudo-instructions .............................................................. 121

Summary ................................................................................................ 121
Descriptions of pseudo-instructions ............................................... 122

Assembler diagnostics ........................................................................ 131

Message format .................................................................................. 131
Severity levels ..................................................................................... 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

Introduction ......................................................................................... 133
Thumb code labels ........................................................................... 133
Alternative register names .............................................................. 134
Alternative mnemonics ................................................................... 135
Operator synonyms ........................................................................ 136
Warning messages ........................................................................... 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.


*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](http://www.arm.com), that contains information and news about the ARM cores.

● The IAR Systems web site, [www.iar.com](http://www.iar.com), that holds application notes and other product information.
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 6.2\arm\doc.

**TYPOGRAPHIC CONVENTIONS**

The IAR Systems documentation set uses the following typographic conventions:

<table>
<thead>
<tr>
<th>Style</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>• Source code examples and file paths.</td>
</tr>
<tr>
<td></td>
<td>• Text on the command line.</td>
</tr>
<tr>
<td></td>
<td>• Binary, hexadecimal, and octal numbers.</td>
</tr>
<tr>
<td>parameter</td>
<td>A placeholder for an actual value used as a parameter, for example filename.h where filename represents the name of the file.</td>
</tr>
<tr>
<td>[option]</td>
<td>An optional part of a command.</td>
</tr>
<tr>
<td>[ab</td>
<td>c]</td>
</tr>
<tr>
<td>(ab</td>
<td>c)</td>
</tr>
<tr>
<td>bold</td>
<td>Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.</td>
</tr>
<tr>
<td>italic</td>
<td>• A cross-reference within this guide or to another guide.</td>
</tr>
<tr>
<td></td>
<td>• Emphasis.</td>
</tr>
<tr>
<td>...</td>
<td>An ellipsis indicates that the previous item can be repeated an arbitrary number of times.</td>
</tr>
</tbody>
</table>

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:

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

Table 2: Naming conventions used in this guide
Document conventions
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:

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

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:

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

Table 4: Assembler error return codes
Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

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

The components are separated by spaces or tabs.

A source line cannot exceed 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).

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 -2,147,483,648 to 2,147,483,647.

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:

<table>
<thead>
<tr>
<th>Integer type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1010b, b'1010</td>
</tr>
<tr>
<td>Octal</td>
<td>1234q, q'1234</td>
</tr>
<tr>
<td>Decimal</td>
<td>1234, -1, d'1234</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>0xFFFFh, 0xFFFF, h'FFFF</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A'B'</td>
<td>A'B</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>Empty string (no value).</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>for double quote within a string</td>
</tr>
</tbody>
</table>

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:

\[ [+|-] (\text{digits}) . (\text{digits}) ((E|e)[+|-] \text{digits}) \]

This table shows some valid examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.23</td>
<td>1.023 \times 10^1</td>
</tr>
<tr>
<td>1.23456E-24</td>
<td>1.23456 \times 10^{-24}</td>
</tr>
<tr>
<td>1.0E3</td>
<td>1.0 \times 10^3</td>
</tr>
</tbody>
</table>

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 \texttt{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 \texttt{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 \texttt{PUBLIC} directive to make one or more symbols available to other modules. The \texttt{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 \textit{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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPSR</td>
<td>32 bits</td>
<td>Current program status register</td>
</tr>
<tr>
<td>D0–D31</td>
<td>64 bits</td>
<td>Floating-point coprocessor registers for double precision</td>
</tr>
<tr>
<td>Q0–Q15</td>
<td>128 bits</td>
<td>Advanced SIMD registers</td>
</tr>
<tr>
<td>FPSCR</td>
<td>32 bits</td>
<td>Floating-point coprocessor, status and control register</td>
</tr>
<tr>
<td>FPSID</td>
<td>32 bits</td>
<td>Floating-point coprocessor, system ID register</td>
</tr>
<tr>
<td>R0–R12</td>
<td>32 bits</td>
<td>General purpose registers</td>
</tr>
<tr>
<td>R13 (SP)</td>
<td>32 bits</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>R14 (LR)</td>
<td>32 bits</td>
<td>Link register</td>
</tr>
<tr>
<td>R15 (PC)</td>
<td>32 bits</td>
<td>Program counter</td>
</tr>
<tr>
<td>S0–S31</td>
<td>32 bits</td>
<td>Floating-point coprocessor registers for single precision</td>
</tr>
<tr>
<td>SPSR</td>
<td>32 bits</td>
<td>Saved program status register</td>
</tr>
</tbody>
</table>

*Table 8: Predefined register symbols*

In addition, specific cores might allow you to use other register symbols, for example APSCR 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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARM_ADVANCED_SIMD</strong></td>
<td>An integer that is set based on the --cpu 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.</td>
</tr>
<tr>
<td><strong>ARM_MEDIA</strong></td>
<td>An integer that is set based on the --cpu 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.</td>
</tr>
<tr>
<td><strong>ARM_MPCORE</strong></td>
<td>An integer that is set based on the --cpu option. The symbol is set to 1 if the selected processor architecture has the Multiprocessing Extensions. The symbol is undefined for other cores.</td>
</tr>
<tr>
<td><strong>ARM_PROFILE_M</strong></td>
<td>An integer that is set based on the --cpu option. The symbol is set to 1 if the selected processor is a profile M core. The symbol is undefined for other cores.</td>
</tr>
<tr>
<td><strong>ARMVFP</strong></td>
<td>An integer that is set based on the --fpu option and that identifies whether floating-point instructions for a vector floating-point coprocessor have been enabled or not. The symbol is defined to <strong>ARMVFPV2</strong>, <strong>ARMVFPV3</strong> or <strong>ARMVFPV4</strong>_. These symbolic names can be used when testing the <strong>ARMVFP</strong> symbol. If floating-point instructions are disabled (default), the symbol is undefined.</td>
</tr>
<tr>
<td><strong>BUILD_NUMBER</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>The current date in dd/Mmm/yyyy format (string).</td>
</tr>
<tr>
<td><strong>FILE</strong></td>
<td>The name of the current source file (string).</td>
</tr>
<tr>
<td><strong>IAR_SYSTEMS_ASM</strong></td>
<td>IAR assembler identifier (number). Note that the number could be higher in a future version of the product. This symbol can be tested with #ifdef to detect whether the code was assembled by an assembler from IAR Systems.</td>
</tr>
<tr>
<td><strong>IASMARM</strong></td>
<td>An integer that is set to 1 when the code is assembled with the IAR Assembler for ARM.</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>The current source line number (number).</td>
</tr>
</tbody>
</table>

Table 9: Predefined symbols
In addition, predefined symbols are defined that allow you to identify the core you are assembling for, for example __ARM5__ and __CORE__. For more 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:

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

Testing symbols for conditional assembly

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

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:

```c
#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:

```assembly
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.
**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:

```c
__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:

```c
UA0_RBR DEFINE 0x40050000
```

**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:

<table>
<thead>
<tr>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>The symbol’s user-defined name.</td>
</tr>
<tr>
<td>Mode</td>
<td>ABS (Absolute), or REL (Relocatable).</td>
</tr>
<tr>
<td>Sections</td>
<td>The name of the section that this symbol is defined relative to.</td>
</tr>
<tr>
<td>Value/Offset</td>
<td>The value (address) of the symbol within the current module, relative to the beginning of the current section.</td>
</tr>
</tbody>
</table>

*Table 10: Symbol and cross-reference table*
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.
Programming hints
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\n
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:

<table>
<thead>
<tr>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-B</td>
<td>Macro execution information</td>
</tr>
<tr>
<td>-c</td>
<td>Conditional list</td>
</tr>
<tr>
<td>--cpu</td>
<td>Core configuration</td>
</tr>
<tr>
<td>-D</td>
<td>Defines preprocessor symbols</td>
</tr>
<tr>
<td>-E</td>
<td>Maximum number of errors</td>
</tr>
<tr>
<td>-e</td>
<td>Generates code in big-endian byte order</td>
</tr>
<tr>
<td>--endian</td>
<td>Specifies the byte order for code and data</td>
</tr>
<tr>
<td>-f</td>
<td>Extends the command line</td>
</tr>
<tr>
<td>--fpu</td>
<td>Floating-point coprocessor architecture configuration</td>
</tr>
<tr>
<td>-G</td>
<td>Opens standard input as source</td>
</tr>
<tr>
<td>-g</td>
<td>Disables the automatic search for system include files</td>
</tr>
<tr>
<td>-I</td>
<td>Adds a search path for a header file</td>
</tr>
<tr>
<td>-i</td>
<td>Lists #included text</td>
</tr>
<tr>
<td>-j</td>
<td>Enables alternative register names, mnemonics, and operators</td>
</tr>
<tr>
<td>-L</td>
<td>Generates a list file to path</td>
</tr>
<tr>
<td>-l</td>
<td>Generates a list file</td>
</tr>
<tr>
<td>--legacy</td>
<td>Generates code linkable with older toolchains.</td>
</tr>
<tr>
<td>-M</td>
<td>Macro quote characters</td>
</tr>
<tr>
<td>-N</td>
<td>Omits header from the assembler listing</td>
</tr>
<tr>
<td>-n</td>
<td>Enables support for multibyte characters</td>
</tr>
<tr>
<td>-O</td>
<td>Sets the object filename to path</td>
</tr>
<tr>
<td>-o</td>
<td>Sets the object filename</td>
</tr>
</tbody>
</table>

*Table 11: Assembler options summary*
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**

```plaintext
-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**
Description of assembler options

\textbf{-c}

Syntax

\texttt{-c(D|M|E|A|O)}

Parameters

- \texttt{D} Disables list file
- \texttt{M} Includes macro definitions
- \texttt{E} Excludes macro expansions
- \texttt{A} Includes assembled lines only
- \texttt{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 \texttt{-L} or \texttt{-l}.

See also

\texttt{-L}, page 41.

To set related options, select:

\texttt{Project>Options>Assembler >List}

\textbf{--cpu}

Syntax

\texttt{--cpu target\_core}

Parameters

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

Description

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

See also

The \textit{IAR C/C++ Development Guide for ARM} for a complete list of coprocessor architecture variants.

\texttt{Project>Options>General Options>Target>Processor variant>Core}
Assembler options

-D

Syntax

-Dsymbol[=value]

Parameters

symbol The name of the symbol you want to define.

value The value of the symbol. If no value is specified, 1 is used.

Description

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

Example

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

```c
#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:

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

-E

Syntax

-Enumber

Parameters

number The number of errors before the assembler stops the assembly.

number 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>Preprocessor>Defined symbols

Project>Options>Assembler>Diagnostics>Max number of errors
Description of assembler options

-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
little, l (default) Specifies little-endian byte order.
big, b 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
filename The commands that you want to extend the command line with are read from the specified file. Notice that there must be a space between the option itself and the filename.
For information about specifying a filename, see 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

**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.

---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**
Description of assembler options

-I

Syntax

-I path

Parameters

path The search path for #include files.

Description

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

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

Example

For example, using the options:

-IC:\global\  -IC:\thisproj\headers\  

and then writing:

#include "asmlib.hdr"

in the source code, make the assembler search first in the current directory, then in the directory C:\global, and then in the directory C:\thisproj\headers. Finally, the assembler searches the directories specified in the IASMARM_INC environment variable, provided that this variable is set, and in the system header directories.

Project>Options>Assembler>Preprocessor>Additional include directories

-i

Syntax

-i

Description

Use this option to list #include files in the list file.

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 -i option allows you to list these file lines.

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**
Operator synonyms, page 51 and the chapter Migrating to the IAR Assembler for ARM.

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

- **-L**

**Syntax**
- L[\path]

**Parameters**
- **No parameter**
  - Generates a listing with the same name as the source file, but with the filename extension .lst.
- **\path**
  - 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 [\path]sourcename.lst.

- 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 \filename

**Parameters**
- **\filename**
  - The output is stored in the specified file. Note that you must include a space before the filename. If no extension is specified, .lst is used.
**Description of assembler options**

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 -N'<>'
```

**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**
Description of assembler options

-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.
Description of assembler options

-**s**

**Syntax**

-**s**(+-)

**Parameters**

+ 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 `LABEL` and `label` refer to different symbols. When `-s` is used, `LABEL` and `label` instead refer to the same symbol.

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

--**system_include_dir**

**Syntax**

--**system_include_dir** path

**Parameters**

path The path to the system include files.

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

**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**

-tn

**Parameters**

n 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**

```-U symbol```

**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. |
| + | Enables all warnings. |
| - | Disables all warnings. |
| +n | Enables just warning n. |
| -n | Disables just warning n. |
| +m-n | Enables warnings m to n. |
| -m-n | Disables warnings m to n. |
Description of assembler options

-w  Generates the exit code 1 if a warning message is produced. By default, warnings generate exit code 0.

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 -w option can only be used once on the command line.

Example
To disable just warning 0 (unreferenced label), use this command:
`iasmarm prog -w-0`
To disable warnings 0 to 8, use this command:
`iasmarm prog -w-0-8`

See also
Assembler diagnostics, page 131.

To set related options, select:
Project>Options>Assembler>Diagnostics

-x

Syntax
-x[D|I|2]

Parameters
D  Includes preprocessor #defines.
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 -L or -l.

See also
-L, page 41.

Project>Options>Assembler>List>Include cross reference
Assembler operators

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

Precedence of 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 – 1

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>Parenthesis</td>
</tr>
</tbody>
</table>

UNARY OPERATORS – 1

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Unary plus</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
</tr>
</tbody>
</table>
Summary of assembler operators

- 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:

<table>
<thead>
<tr>
<th>Operator synonym</th>
<th>Precedence</th>
<th>Operator</th>
<th>Precedence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>:AND:</td>
<td>3</td>
<td>&amp;</td>
<td>5</td>
<td>Bitwise AND.</td>
</tr>
<tr>
<td>:EOR:</td>
<td>3</td>
<td>^</td>
<td>6</td>
<td>Bitwise exclusive OR.</td>
</tr>
<tr>
<td>:LAND:</td>
<td>8</td>
<td>&amp;&amp;</td>
<td>5</td>
<td>Logical AND.</td>
</tr>
<tr>
<td>:LEOR:</td>
<td>8</td>
<td>XOR</td>
<td>6</td>
<td>Logical exclusive OR.</td>
</tr>
<tr>
<td>:LNOT:</td>
<td>1</td>
<td>!</td>
<td>1</td>
<td>Logical NOT.</td>
</tr>
<tr>
<td>:lor:</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:MOD:</td>
<td>2</td>
<td>%</td>
<td>2</td>
<td>Modulo.</td>
</tr>
</tbody>
</table>

*Table 12: Operator synonyms*
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**

\[
1 + 2 \times 3 \rightarrow 7 \\
(1 + 2) \times 3 \rightarrow 9
\]

### * 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**

\[
2 \times 2 \rightarrow 4 \\
-2 \times 2 \rightarrow -4
\]

---

**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*.
+ **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.
Description of assembler operators

Example

\[
\begin{align*}
9/2 & \rightarrow 4 \\
-12/3 & \rightarrow -4 \\
9/2*6 & \rightarrow 24
\end{align*}
\]

### < 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**

\[
\begin{align*}
-1 & < 2 \rightarrow 1 \\
2 & < 1 \rightarrow 0 \\
2 & < 2 \rightarrow 0
\end{align*}
\]

### <= 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**

\[
\begin{align*}
1 & \leq 2 \rightarrow 1 \\
2 & \leq 1 \rightarrow 0 \\
1 & \leq 1 \rightarrow 1
\end{align*}
\]

### <>, != 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**

\[
\begin{align*}
1 & <> 2 \rightarrow 1 \\
2 & <> 2 \rightarrow 0 \\
'A' & <> 'B' \rightarrow 1
\end{align*}
\]

### =, == 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**

\[
\begin{align*}
1 & = 2 \rightarrow 0 \\
2 & == 2 \rightarrow 1 \\
'\text{ABC}' & = '\text{ABCD}' \rightarrow 0
\end{align*}
\]
> 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 \(\rightarrow\) 0
2 > 1 \(\rightarrow\) 1
1 > 1 \(\rightarrow\) 0

>= Greater than or equal (7)

Description

\(\geq\) 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 \(\geq\) 2 \(\rightarrow\) 0
2 \(\geq\) 1 \(\rightarrow\) 1
1 \(\geq\) 1 \(\rightarrow\) 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 \(\rightarrow\) 1
1010B && 0101B \(\rightarrow\) 1
1010B && 0000B \(\rightarrow\) 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 \(\rightarrow\) 0010B
1010B & 0101B \(\rightarrow\) 0000B
1010B & 0000B \(\rightarrow\) 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 → 11111111111111111111111111110101B

| 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.

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.

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.

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 → 1110000B
0000011111111111B << 5 → 1111111111110000B
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
1111111111111111B >> 20 → 0
14 >> 1 → 7
BYTE1 First byte (1)

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

\[
\text{BYTE1 } 0x\text{ABCD} \rightarrow 0xCD
\]

BYTE2 Second byte (1)

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

\[
\text{BYTE2 } 0x12345678 \rightarrow 0x56
\]

BYTE3 Third byte (1)

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

\[
\text{BYTE3 } 0x12345678 \rightarrow 0x34
\]

BYTE4 Fourth byte (1)

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

\[
\text{BYTE4 } 0x12345678 \rightarrow 0x12
\]

DATE Current time/date (1)

Description

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

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

\[
\begin{align*}
\text{DATE 1} & \quad \text{Current second (0–59).} \\
\text{DATE 2} & \quad \text{Current minute (0–59).} \\
\text{DATE 3} & \quad \text{Current hour (0–23).} \\
\text{DATE 4} & \quad \text{Current day (1–31).}
\end{align*}
\]
Example

To assemble the date of assembly:

today: DC8 DATE 5, DATE 4, DATE 3

**HIGH** High byte (1)

**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 (1)

**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 (1)

**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 (1)

**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 (1)

**Syntax**

SFB(section [{+|-}offset])
Description of assembler operators

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>section</td>
<td>The name of a section, which must be defined before SFB is used.</td>
</tr>
<tr>
<td>offset</td>
<td>An optional offset from the start address. The parentheses are optional if offset is omitted.</td>
</tr>
</tbody>
</table>

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

```assembly
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 (1)

Syntax

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>section</td>
<td>The name of a section, which must be defined before SFE is used.</td>
</tr>
<tr>
<td>offset</td>
<td>An optional offset from the start address. The parentheses are optional if offset is omitted.</td>
</tr>
</tbody>
</table>

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

```assembly
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 (1)**

**Syntax**

```
SIZEOF section
```

**Parameters**

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

**Description**

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

**Example**

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

**Table.s**:

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

**Application.s**:

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

**UGT Unsigned greater than (7)**

**Description**

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

**Example**

```
2 UGT 1 → 1
-1 UGT 1 → 1
```

**ULT Unsigned less than (7)**

**Description**

ULT evaluates to 1 (true) if the left operand has a smaller value than the right operand, otherwise it is 0 (false). The operation treats the operands as unsigned values.
XOR Logical exclusive OR (6)

Description
XOR or the synonym \texttt{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 \texttt{XOR} to perform logical XOR on its two operands.

Note: The precedence of \texttt{LEOR} is 8.

Example
\begin{align*}
0101\text{B} & \text{ XOR } 1010\text{B} \rightarrow 0 \\
0101\text{B} & \text{ XOR } 0000\text{B} \rightarrow 1
\end{align*}
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:

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

Table 13: Assembler directives summary
Summary of assembler directives

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

Table 13: Assembler directives summary (Continued)
<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC16</td>
<td>Generates 16-bit constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DC24</td>
<td>Generates 24-bit constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DC32</td>
<td>Generates 32-bit constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>D8B</td>
<td>Generates 8-bit byte constants, including strings.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DCD</td>
<td>Generates 32-bit long word constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DCW</td>
<td>Generates 16-bit word constants, including strings.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DEFINE</td>
<td>Defines a file-wide value.</td>
<td>Value assignment</td>
</tr>
<tr>
<td>DF32</td>
<td>Generates 32-bit floating-point constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DF64</td>
<td>Generates 64-bit floating-point constants.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DS8</td>
<td>Allocates space for 8-bit integers.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DS16</td>
<td>Allocates space for 16-bit integers.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DS24</td>
<td>Allocates space for 24-bit integers.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>DS32</td>
<td>Allocates space for 32-bit integers.</td>
<td>Data definition or allocation</td>
</tr>
<tr>
<td>ELSE</td>
<td>Assembles instructions if a condition is false.</td>
<td>Conditional assembly</td>
</tr>
<tr>
<td>ELSEIF</td>
<td>Specifies a new condition in an IF...ENDIF block.</td>
<td>Conditional assembly</td>
</tr>
<tr>
<td>END</td>
<td>Ends the assembly of the last module in a file.</td>
<td>Module control</td>
</tr>
<tr>
<td>ENDM</td>
<td>Ends a macro definition.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>ENDR</td>
<td>Ends a repeat structure.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>EQU</td>
<td>Assigns a permanent value local to a module.</td>
<td>Value assignment</td>
</tr>
</tbody>
</table>

Table 13: Assembler directives summary (Continued)
Summary of assembler directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEN</td>
<td>Aligns the program counter to an even address.</td>
<td>Section control</td>
</tr>
<tr>
<td>EXITM</td>
<td>Exits prematurely from a macro.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>EXTERN</td>
<td>Imports an external symbol.</td>
<td>Symbol control</td>
</tr>
<tr>
<td>EXTWEAK</td>
<td>Imports an external symbol; the symbol may be undefined.</td>
<td>Symbol control</td>
</tr>
<tr>
<td>IF</td>
<td>Assembles instructions if a condition is true.</td>
<td>Conditional assembly</td>
</tr>
<tr>
<td>IMPORT</td>
<td>Imports an external symbol.</td>
<td>Symbol control</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>Includes a file.</td>
<td>Assembler control</td>
</tr>
<tr>
<td>LIBRARY</td>
<td>Begins a module; an alias for PROGRAM and NAME.</td>
<td>Module control</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Creates symbols local to a macro.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>LSTCND</td>
<td>Controls conditional assembler listing.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTCOD</td>
<td>Controls multi-line code listing.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTEXP</td>
<td>Controls the listing of macro generated lines.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTMAC</td>
<td>Controls the listing of macro definitions.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTOUT</td>
<td>Controls assembler-listing output.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTPAG</td>
<td>Retained for backward compatibility reasons. Recognized but ignored.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTREP</td>
<td>Controls the listing of lines generated by repeat directives.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTSAS</td>
<td>Controls structured assembly listing.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LSTXRF</td>
<td>Generates a cross-reference table.</td>
<td>Listing control</td>
</tr>
<tr>
<td>LTORG</td>
<td>Directs the current literal pool to be assembled immediately following the directive.</td>
<td>Assembler control</td>
</tr>
<tr>
<td>MACRO</td>
<td>Defines a macro.</td>
<td>Macro processing</td>
</tr>
<tr>
<td>MODULE</td>
<td>Begins a module; an alias for PROGRAM and NAME.</td>
<td>Module control</td>
</tr>
<tr>
<td>NAME</td>
<td>Begins a program module.</td>
<td>Module control</td>
</tr>
<tr>
<td>ODD</td>
<td>Aligns the program location counter to an odd address.</td>
<td>Section control</td>
</tr>
<tr>
<td>OVERLAY</td>
<td>Recognized but ignored.</td>
<td>Symbol control</td>
</tr>
<tr>
<td>PAGE</td>
<td>Retained for backward compatibility reasons.</td>
<td>Listing control</td>
</tr>
<tr>
<td>PAGESIZ</td>
<td>Retained for backward compatibility reasons.</td>
<td>Listing control</td>
</tr>
<tr>
<td>PRESERVE8</td>
<td>Sets a module attribute.</td>
<td>Module control</td>
</tr>
</tbody>
</table>

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.

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

Table 14: Module control directives
Module control directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Expression restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Begins a module; alias to PROGRAM.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>PRESERVE8</td>
<td>Sets a module attribute that informs the linker that all exported functions in the module preserves an 8-byte aligned stack.</td>
<td>The assembler does not verify that the claims are fulfilled.</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>Begins a module.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>REQUIRE8</td>
<td>Sets a module attribute that informs the linker that the module requires an 8-byte aligned stack.</td>
<td></td>
</tr>
<tr>
<td>RTMODEL</td>
<td>Declares runtime model attributes.</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

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**.
Symbol control directives

These directives control how symbols are shared between modules:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTERN, IMPORT</td>
<td>Imports an external symbol.</td>
</tr>
<tr>
<td>EXTWEAK</td>
<td>Imports an external symbol; the symbol may be undefined.</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>Exports symbols to other modules.</td>
</tr>
<tr>
<td>PUBWEAK</td>
<td>Exports symbols to other modules, multiple definitions allowed.</td>
</tr>
<tr>
<td>REQUIRE</td>
<td>Forces a symbol to be referenced.</td>
</tr>
</tbody>
</table>

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.

```assembly
    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:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM, CODE32</td>
<td>Subsequent instructions are assembled as 32-bit (ARM) instructions. Labels within a CODE32 area have bit 0 set to 0. Force 4-byte alignment.</td>
</tr>
<tr>
<td>CODE16</td>
<td>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.</td>
</tr>
<tr>
<td>DATA</td>
<td>Defines an area of data within a code section, where labels work as in a CODE32 area.</td>
</tr>
<tr>
<td>THUMB</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

```assembly
name    modeChange
section MYCODE:CODE(2)
thumb

thumbEntry
  bx      pc              ; Branch to armEntry, and
  nop     ; change execution mode.
  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.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Expression restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGNRAM</td>
<td>Aligns the program location counter by incrementing it.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>ALIGNROM</td>
<td>Aligns the program location counter by inserting zero-filled bytes.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>EVEN</td>
<td>Aligns the program counter to an even address.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>ODD</td>
<td>Aligns the program counter to an odd address.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>RSEG</td>
<td>Begins an ELF section; alias to SECTION.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>SECTION</td>
<td>Begins an ELF section.</td>
<td>No external references Absolute</td>
</tr>
<tr>
<td>SECTION_TYPE</td>
<td>Sets ELF type and flags for a section.</td>
<td></td>
</tr>
</tbody>
</table>

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
align 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.
flag ROOT, NORoot
ROOT (the default mode) indicates that the section fragment must not be discarded.
NORoot means that the section fragment is discarded by the linker if no symbols in this section fragment are referred to. Normally, all section fragments except startup code and interrupt vectors should set this flag.
REORDER, NOreORDER
NOreORDER (the default mode) starts a new fragment in the section with the given name, or a new section if no such section exists. REORDER starts a new section with the given name.
section The name of the section. The section name is a user-defined symbol that follows the rules described in Symbols, page 24.
type The memory type, which can be either CODE, CONST, or DATA.
value Byte value used for padding, default is zero.
type-expr A constant expression identifying the ELF type of the section.
flags-expr A constant expression identifying the ELF flags of the section.
Section control directives

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:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=, EQU</td>
<td>Assigns a permanent value local to a module.</td>
</tr>
<tr>
<td>ALIAS</td>
<td>Assigns a permanent value local to a module.</td>
</tr>
<tr>
<td>ASSIGN, SET, SETA, VAR</td>
<td>Assigns a temporary value.</td>
</tr>
<tr>
<td>DEFINE</td>
<td>Defines a file-wide value.</td>
</tr>
</tbody>
</table>

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

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

DESCRIPTIONS

Defining a temporary value

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

Defining a permanent local value

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

**Defining a permanent global value**

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

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

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

**EXAMPLES**

**Redefining a symbol**

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

```assembly
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:

```
 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
```
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.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Expression restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELSE</td>
<td>Assembles instructions if a condition is false.</td>
<td>No forward references</td>
</tr>
<tr>
<td>ELSEIF</td>
<td>Specifies a new condition in an IF...ENDIF block.</td>
<td>No external references Absolute Fixed</td>
</tr>
<tr>
<td>ENDIF</td>
<td>Ends an IF block.</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>Assembles instructions if a condition is true.</td>
<td>No forward references No external references Absolute Fixed</td>
</tr>
</tbody>
</table>

Table 19: Conditional assembly directives
SYNTAX
ELSE
ELSEIF condition
ENDIF
IF condition

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

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

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

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

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

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

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

Table 20: Macro processing directives

SYNTAX

_begin _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
LOCAL symbol [,symbol] ...

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

REPT expr
REPTC formal,actual
REPTI formal,actual [,actual] ...

PARAMETERS

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

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
```
Macro processing directives

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:

```
name errMacro
extern abort
errMac 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:

```assembly
name    cmpMacro
cmp_reg     macro   op
    CMP     op
endm
```

The macro can be called using the macro quote characters:

```assembly
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:

```assembly
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 module filler
20 section .text:CODE(2)
21 fill    3
21.1 if      _args == 2
21.2 rept
21.3 dc8     3
21.4 endr
21.5 else
21 00000000 03 fill    3
21.1 endif
21.2 endm
```
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 REPTI to assemble a block of instructions once for each string in a series of strings. Strings containing commas should be enclosed in quotation marks.

**EXAMPLES**

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

**Coding inline for efficiency**

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

This example outputs bytes from a buffer to a port:

```assembly
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

main        bl      play
done        b       done
```

For efficiency we can recode this using a macro:

```assembly
play        macro   buf,size,port
            local   loop
            ldr     r1,=buf         ; Pointer to buffer.
            ldr     r2,=port        ; Pointer to ioPort.
            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.
            end
```

```assembly
name    ioBufferSubroutine
section MYCODE:CODE(2)

name    ioBufferInline
section MYCODE:CODE(2)
```
Macro processing directives

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"
            move    r0,#'chr'       ; Pass char as parameter.
            bl      plotc
endr

This produces this code:

9                          name    reptc
10                          extern  plotc
11                          section MYCODE:CODE(2)
12
13                          banner   reptc   chr,"Welcome"
This example uses **REPTI** to clear several memory locations:

```
This produces this code:

```
### Listing control directives

These directives provide control over the assembler list file:

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

**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:

```assembly
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:

```assembly
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:

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>name lstcndTest</td>
</tr>
<tr>
<td>10</td>
<td>extern print</td>
</tr>
<tr>
<td>11</td>
<td>section FLASH:CODE(2)</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>debug set 0</td>
</tr>
<tr>
<td>14</td>
<td>begin if debug</td>
</tr>
<tr>
<td>15</td>
<td>bl print</td>
</tr>
<tr>
<td>16</td>
<td>endif</td>
</tr>
<tr>
<td>17</td>
<td>lstcnd+</td>
</tr>
<tr>
<td>18</td>
<td>begin2 if debug</td>
</tr>
<tr>
<td>19</td>
<td>endif</td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

**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-
```
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:

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

Table 22: C-style preprocessor directives
Table 22: C-style preprocessor directives  (Continued)

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

SYNTAX

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

PARAMETERS

condition | An absolute expression | The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true.

filename   | Name of file to be included or referred. |
line-no    | Source line number. |
message    | Text to be displayed. |
C-style preprocessor directives

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.

module misplacedComment1
expression equ x * 8 + 5
;...
end
```
This example illustrates some problems that might occur when assembler comments are used in the C-style preprocessor:

```c
#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.

```c
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
```
In this 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"

xchReg 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.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC8</td>
<td>DCB</td>
<td>Generates 8-bit constants, including strings.</td>
</tr>
</tbody>
</table>

Table 23: Data definition or allocation directives
Data definition or allocation directives

### SYNTAX

<table>
<thead>
<tr>
<th>Directive</th>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC16</td>
<td>DCFW</td>
<td>Generates 16-bit constants.</td>
</tr>
<tr>
<td>DC24</td>
<td>DCW</td>
<td>Generates 24-bit constants.</td>
</tr>
<tr>
<td>DC32</td>
<td>DCD</td>
<td>Generates 32-bit constants.</td>
</tr>
<tr>
<td>DF32</td>
<td>DCF32</td>
<td>Generates 32-bit floating-point constants.</td>
</tr>
<tr>
<td>DF64</td>
<td>DCF64</td>
<td>Generates 64-bit floating-point constants.</td>
</tr>
<tr>
<td>DS8</td>
<td>DS</td>
<td>Allocates space for 8-bit integers.</td>
</tr>
<tr>
<td>DS16</td>
<td>DS</td>
<td>Allocates space for 16-bit integers.</td>
</tr>
<tr>
<td>DS24</td>
<td>DS</td>
<td>Allocates space for 24-bit integers.</td>
</tr>
<tr>
<td>DS32</td>
<td>DS</td>
<td>Allocates space for 32-bit integers.</td>
</tr>
</tbody>
</table>

### 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, DC32, 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.

```assembly
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:

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

To define a string which includes a trailing zero:

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

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

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

Reserving space

To reserve space for 10 bytes:

```asm
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.

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

*Table 24: Assembler control directives*
SYNTAX

FILENAME
/*COMMENT*/
//COMMENT
CASEOFF
CASEON
INCLUDE filename
LTORG
RADIX expr

PARAMETERS

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

```assembly
; 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:

```assembly
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:

```assembly
/*
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:

```assembly
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:

```assembly
module caseSensitivity1
section MYCODE:CODE(2)
caseoff
label       nop                 ; Stored as "LABEL".
b       LABEL
end
```

The following will generate a duplicate label error:

```assembly
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.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI BASEADDRESS</td>
<td>Declares a base address CFA (Canonical Frame Address).</td>
</tr>
<tr>
<td>CFI BLOCK</td>
<td>Starts a data block.</td>
</tr>
<tr>
<td>CFI CODEALIGN</td>
<td>Declares code alignment.</td>
</tr>
<tr>
<td>CFI COMMON</td>
<td>Starts or extends a common block.</td>
</tr>
<tr>
<td>CFI CONDITIONAL</td>
<td>Declares a data block to be a conditional thread.</td>
</tr>
<tr>
<td>CFI DATAALIGN</td>
<td>Declares data alignment.</td>
</tr>
<tr>
<td>CFI DEFAULT</td>
<td>Declares the default state of all resources.</td>
</tr>
<tr>
<td>CFI ENDBLOCK</td>
<td>Ends a data block.</td>
</tr>
<tr>
<td>CFI ENDCOMMON</td>
<td>Ends a common block.</td>
</tr>
<tr>
<td>CFI ENDNAMES</td>
<td>Ends a names block.</td>
</tr>
<tr>
<td>CFI FRAMECALL</td>
<td>Creates a reference into the caller’s frame.</td>
</tr>
<tr>
<td>CFI FUNCALL</td>
<td>Declares function calls for stack usage analysis.</td>
</tr>
<tr>
<td>CFI FUNCTION</td>
<td>Declares a function associated with a data block.</td>
</tr>
<tr>
<td>CFI INDIRECTCALL</td>
<td>Declares indirect calls for stack usage analysis.</td>
</tr>
<tr>
<td>CFI INVALID</td>
<td>Starts a range of invalid backtrace information.</td>
</tr>
<tr>
<td>CFI NAMES</td>
<td>Starts a names block.</td>
</tr>
<tr>
<td>CFI NOCALLS</td>
<td>Declares absence of calls for stack usage analysis.</td>
</tr>
<tr>
<td>CFI NOFUNCTION</td>
<td>Declares a data block to not be associated with a function.</td>
</tr>
<tr>
<td>CFI PICKER</td>
<td>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.</td>
</tr>
<tr>
<td>CFI REMEMBERSSTATE</td>
<td>Remembers the backtrace information state.</td>
</tr>
<tr>
<td>CFI RESOURCE</td>
<td>Declares a resource.</td>
</tr>
<tr>
<td>CFI RESTORESTATE</td>
<td>Restores the saved backtrace information state.</td>
</tr>
<tr>
<td>CFI RETURNADDRESS</td>
<td>Declares a return address column.</td>
</tr>
<tr>
<td>CFI STACKFRAME</td>
<td>Declares a stack frame CFA.</td>
</tr>
<tr>
<td>CFI VALID</td>
<td>Ends a range of invalid backtrace information.</td>
</tr>
<tr>
<td>CFI cfa</td>
<td>Declares the value of a CFA.</td>
</tr>
<tr>
<td>CFI resource</td>
<td>Declares the value of a resource.</td>
</tr>
</tbody>
</table>

*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\} }


Call frame information directives

**CFI resource cfiexpr**

**Stack usage analysis directives**

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

**PARAMETERS**

- **bits**
  - The size of the resource in bits.

- **callee**
  - The label of the called function.

- **caller**
  - The label of the calling function.

- **cfa**
  - The name of a CFA (canonical frame address).

- **cfiexpr**
  - A CFI expression (see *Using expressions for complex cases*, page 114).

- **codealignfactor**
  - The smallest factor of all instruction sizes. Each CFI directive for a data block must be placed according to this alignment. 1 is the default and can always be used, but a larger value shrinks the produced backtrace information in size. The possible range is 1–256.

- **commonblock**
  - The name of a previously defined common block.

- **constant**
  - A constant value or an assembler expression that can be evaluated to a constant value.

- **dataalignfactor**
  - The smallest factor of all frame sizes. If the stack grows toward higher addresses, the factor is negative; if it grows toward lower addresses, the factor is positive. 1 is the default, but a larger value shrinks the produced backtrace information in size. The possible ranges are -256 to -1 and 1 to 256.

- **label**
  - A function label.

- **name**
  - The name of the block.

- **namesblock**
  - The name of a previously defined names block.

- **offset**
  - The offset relative the CFA. An integer with an optional sign.

- **part**
  - A part of a composite resource. The name of a previously declared resource.
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.
Call frame information directives

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:
  \[\text{CFI BASEADDRESS cfa type}\]
  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:

\[\text{CFI COMMON name USING namesblock}\]

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:

\[\text{CFI RETURNADDRESS resource type}\]

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:

\[\text{CFI ENDCOMMON name}\]

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:

\[\text{CFI BLOCK name USING commonblock}\]
Call frame information directives

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
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
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.
## Assembler directives

### Unary operators

**Overall syntax:** \texttt{OPERATOR(operand)}

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

*Table 26: Unary operators in CFI expressions*

### Binary operators

**Overall syntax:** \texttt{OPERATOR(operand1,operand2)}

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>cfiexpr,cfiexpr</td>
<td>Addition</td>
</tr>
<tr>
<td>AND</td>
<td>cfiexpr,cfiexpr</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>DIV</td>
<td>cfiexpr,cfiexpr</td>
<td>Division</td>
</tr>
<tr>
<td>EQ</td>
<td>cfiexpr,cfiexpr</td>
<td>Equal</td>
</tr>
<tr>
<td>GE</td>
<td>cfiexpr,cfiexpr</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>GT</td>
<td>cfiexpr,cfiexpr</td>
<td>Greater than</td>
</tr>
<tr>
<td>LE</td>
<td>cfiexpr,cfiexpr</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>LSHIFT</td>
<td>cfiexpr,cfiexpr</td>
<td>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.</td>
</tr>
<tr>
<td>LT</td>
<td>cfiexpr,cfiexpr</td>
<td>Less than</td>
</tr>
<tr>
<td>MOD</td>
<td>cfiexpr,cfiexpr</td>
<td>Modulo</td>
</tr>
<tr>
<td>MUL</td>
<td>cfiexpr,cfiexpr</td>
<td>Multiplication</td>
</tr>
<tr>
<td>NE</td>
<td>cfiexpr,cfiexpr</td>
<td>Not equal</td>
</tr>
<tr>
<td>OR</td>
<td>cfiexpr,cfiexpr</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>RSHIPTA</td>
<td>cfiexpr,cfiexpr</td>
<td>Arithmetic shift right of the left operand. The number of bits to shift is specified by the right operand. In contrast with RSHIPTL, the sign bit is preserved when shifting.</td>
</tr>
</tbody>
</table>

*Table 27: Binary operators in CFI expressions*
Ternary operators

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
</table>
| FRAME    | cfa, size, offset | Gets the value from a stack frame. The operands are:
- cfa | An identifier denoting a previously declared CFA.
- size | A constant expression denoting a size in bytes.
- offset | A constant expression denoting an offset in bytes.

  Gets the value at address `cfa+offset` of size `size`.

| IF       | cond, true, false | Conditional operator. The operands are:
- cond | A CFA expression denoting a condition.
- true | Any CFA expression.
- false | Any CFA expression.

  If the conditional expression is non-zero, the result is the value of the `true` expression; otherwise the result is the value of the `false` expression.

| LOAD     | size, type, addr  | Gets the value from memory. The operands are:
- size | A constant expression denoting a size in bytes.
- type | A memory type.
- addr | A CFA expression denoting a memory address.

  Gets the value at address `addr` in the memory type `type` of size `size`.

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.

<table>
<thead>
<tr>
<th>Address</th>
<th>CFA</th>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4-R11</th>
<th>R12</th>
<th>R13</th>
<th>R14</th>
<th>Assembler code</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>R13 + 0</td>
<td>—</td>
<td>SAME</td>
<td>—</td>
<td>SAME</td>
<td>—</td>
<td>SAME</td>
<td>—</td>
<td>POP</td>
<td>(r1, 1r)</td>
</tr>
<tr>
<td>00000002</td>
<td>R13 + 8</td>
<td>CFA</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MOVS</td>
<td>r1, #4</td>
</tr>
<tr>
<td>00000004</td>
<td></td>
<td>BL</td>
<td>func2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000008</td>
<td></td>
<td>POP</td>
<td>(r0, lr)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000C</td>
<td>R13 + 0</td>
<td>R0</td>
<td></td>
<td>SAME</td>
<td></td>
<td></td>
<td></td>
<td>MOV</td>
<td>r1, r0</td>
<td></td>
</tr>
<tr>
<td>0000000E</td>
<td>SAME</td>
<td>BX</td>
<td>1r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 MOVS 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

; Stack frame

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

; End names block

cfi    endnames ArmCore
```

**Defining the common block**

```
cfi    common trivialCommon using ArmCore

cfi    codealign 2

cfi    dataalign 4

; CFA

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)
```
Assembler directives

```assembly
.cfi r14 frame(cfa, -4)
cfi cfa rl3+8
movs r1,#4
cfi funcall func2
bl func2
pop {r0,lr}
cfi rl r0
cfi rl4 samevalue
cfi cfa rl3
mov r1,r0
cfi rl 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.
Call frame information directives
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:

<table>
<thead>
<tr>
<th>Pseudo-instruction</th>
<th>Directive</th>
<th>Translated to</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
<td>ARM</td>
<td>ADD, SUB</td>
<td>Loads a program-relative address into a register.</td>
</tr>
<tr>
<td>ADR</td>
<td>CODE16*</td>
<td>ADD</td>
<td>Loads a program-relative address into a register.</td>
</tr>
<tr>
<td>ADR</td>
<td>THUMB</td>
<td>ADD, SUB</td>
<td>Loads a program-relative address into a register.</td>
</tr>
<tr>
<td>ADRL</td>
<td>ARM</td>
<td>ADD, SUB</td>
<td>Loads a program-relative address into a register.</td>
</tr>
<tr>
<td>ADRL</td>
<td>THUMB</td>
<td>ADD, SUB</td>
<td>Loads a program-relative address into a register.</td>
</tr>
<tr>
<td>LDR</td>
<td>ARM</td>
<td>MOV, MVN, LDR</td>
<td>Loads a register with any 32-bit expression.</td>
</tr>
<tr>
<td>LDR</td>
<td>CODE16*</td>
<td>MOV, LDR</td>
<td>Loads a register with any 32-bit expression.</td>
</tr>
<tr>
<td>LDR</td>
<td>THUMB</td>
<td>MOV, MVN, LDR</td>
<td>Loads a register with any 32-bit expression.</td>
</tr>
</tbody>
</table>

Table 30: Pseudo-instructions
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
```

---

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

*Deprecated. Use THUMB instead.*
thumb
thumbLabel ; ...  
end

ADR (CODE16)

Syntax
ADR register, expression

Parameters
register The register to load.
expression A program-relative expression that evaluates to a word-aligned address within the range +4 to +1024 bytes.

Description
This Thumb-1 ADR can generate word-aligned addresses only (that is, addresses divisible by 4). Use the ALIGNROM directive to ensure that the address is aligned (unless DC32 is used, because it is always word-aligned).

ADR (THUMB)

Syntax
ADR{condition} register, expression

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

Description
Similar to ADR (CODE16), but the address range can be larger if a 32-bit Thumb-2 instruction is available in the architecture used.

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

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

Example

name thumbAddr
section MYCODE:CODE(2)
thumb
Descriptions of pseudo-instructions

```
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
```

IAR Assembler

124 Reference Guide for ARM
ADRL (THUMB)

Syntax
ADRL{condition} register,expression

Parameters
(condition) An optional condition code if the instruction is placed after an IT instruction.
register The register to load.
expression A program-relative expression that evaluates to an address within the range ±1 Mbyte.

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

LDR (ARM)

Syntax
LDR{condition} register,=expression1
or
LDR{condition} register,expression2

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

Description
The first form of the LDR pseudo-instruction loads a register with any 32-bit expression. The second form of the instruction reads a 32-bit value from an address specified by the expression.

If the value of expression1 is within the range of a MOV or MVN instruction, the assembler generates the appropriate instruction. If the value of expression1 is not within the range of a MOV or MVN instruction, or if the expression1 is unsolved, the assembler places the constant in a literal pool and generates a program-relative LDR instruction that reads the constant from the literal pool. The offset from the program location counter to the constant must be less than 4 Kbytes.
Descriptions of pseudo-instructions

Example

```
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 0xPFEEDDCC into r2.
  data
     label       dc32    0xFFEEDDCC
  ltorg                   ; The literal pool is placed
       ; here.
end
```

See also

The LTORG directive in the section Assembler control directives, page 102.

**LDR (CODE16)**

**Syntax**

```
LDR register,=expression1
or
LDR register, expression2
```

**Parameters**

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

**Description**

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

**LDR (THUMB)**

**Syntax**

```
LDR{condition} register=expression
```

**Parameters**

- `condition` An optional condition code if the instruction is placed after an IT instruction.
**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**

```plaintext
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.
**Descriptions of pseudo-instructions**

**MOV (CODE16)**

**Syntax**

MOV Rd, Rs

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
<td>The destination register.</td>
</tr>
<tr>
<td>Rs</td>
<td>The source register.</td>
</tr>
</tbody>
</table>

**Description**

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

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

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

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

**Example**

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

**MOV32 (THUMB)**

**Syntax**

MOV32(condition) register, expression

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>An optional condition code if the instruction is placed after an IT instruction.</td>
</tr>
<tr>
<td>register</td>
<td>The register to load.</td>
</tr>
<tr>
<td>expression</td>
<td>Any 32-bit expression.</td>
</tr>
</tbody>
</table>

**Description**

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

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

**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).
Descriptions of pseudo-instructions
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
  4---^  "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**

```plaintext
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-.8)     :AND: -1
```

```plaintext
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:

<table>
<thead>
<tr>
<th>Alternative register name</th>
<th>Assembler register name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>R0</td>
</tr>
<tr>
<td>A2</td>
<td>R1</td>
</tr>
<tr>
<td>A3</td>
<td>R2</td>
</tr>
<tr>
<td>A4</td>
<td>R3</td>
</tr>
<tr>
<td>V1</td>
<td>R4</td>
</tr>
<tr>
<td>V2</td>
<td>R5</td>
</tr>
<tr>
<td>V3</td>
<td>R6</td>
</tr>
<tr>
<td>V4</td>
<td>R7</td>
</tr>
<tr>
<td>V5</td>
<td>R8</td>
</tr>
<tr>
<td>V6</td>
<td>R9</td>
</tr>
<tr>
<td>V7</td>
<td>R10</td>
</tr>
<tr>
<td>SB</td>
<td>R9</td>
</tr>
</tbody>
</table>

*Table 31: Alternative register names*
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:

<table>
<thead>
<tr>
<th>Alternative mnemonic</th>
<th>Assembler mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCS</td>
<td>ADC</td>
</tr>
<tr>
<td>ADDS</td>
<td>ADD</td>
</tr>
<tr>
<td>ANDS</td>
<td>AND</td>
</tr>
<tr>
<td>ASLS</td>
<td>LSL</td>
</tr>
<tr>
<td>ASRS</td>
<td>ASR</td>
</tr>
<tr>
<td>BICS</td>
<td>BIC</td>
</tr>
<tr>
<td>BNCC</td>
<td>BCS</td>
</tr>
<tr>
<td>BNCS</td>
<td>BCC</td>
</tr>
<tr>
<td>BNEQ</td>
<td>BNE</td>
</tr>
<tr>
<td>BNGE</td>
<td>BLT</td>
</tr>
<tr>
<td>BNGT</td>
<td>BLE</td>
</tr>
<tr>
<td>BNHI</td>
<td>BLS</td>
</tr>
<tr>
<td>BNLE</td>
<td>BOT</td>
</tr>
<tr>
<td>BNLO</td>
<td>BCS</td>
</tr>
<tr>
<td>BNLS</td>
<td>BHI</td>
</tr>
<tr>
<td>BNLT</td>
<td>BGE</td>
</tr>
<tr>
<td>BNMI</td>
<td>BPL</td>
</tr>
<tr>
<td>BNNE</td>
<td>BEQ</td>
</tr>
<tr>
<td>BNPL</td>
<td>BMI</td>
</tr>
<tr>
<td>BNVC</td>
<td>BVS</td>
</tr>
</tbody>
</table>

Table 32: Alternative mnemonics

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

<table>
<thead>
<tr>
<th>Operator synonym</th>
<th>Assembler operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>:AND:</td>
<td>&amp;</td>
</tr>
<tr>
<td>:EOR:</td>
<td>^</td>
</tr>
<tr>
<td>:LAND:</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>:LEOR:</td>
<td>XOR</td>
</tr>
<tr>
<td>:LNOT:</td>
<td>!</td>
</tr>
<tr>
<td>:LOR:</td>
<td></td>
</tr>
<tr>
<td>:MOD:</td>
<td>%</td>
</tr>
</tbody>
</table>

**Table 33: Operator synonyms**
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:

\[
\text{MUL } R0, R0, R1
\]

Example of correct code:

\[
\text{MUL } R0, R1
\]

**Immediate \#0 omitted in Load/Store**

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

Example of incorrect code:

\[
\text{LDR } R0, [R1]
\]

Example of correct code:

\[
\text{LDR } R0, [R1, #0]
\]
Warning messages
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
__ARMVFPI (predefined symbol) ........................................ 26
__ARM_ADVANCED_SIMD_ (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

139

AARM-9
assembly messages format ........................................... 131
assembly error messages ........................................... 132
assembler symbols ................................................... 24
assembler source format ............................................. 21
assembler source files, including .................................. 97, 104
assembler source code, porting .................................... 73
assembler options ..................................................... 49
assembler operators ................................................... 49
in expressions .......................................................... 22
precedence .............................................................. 49
assembler output, including debug information .................. 45
assembler pseudo-instructions ...................................... 121
assembler macros ...................................................... 85
arguments, passing to ................................................. 85
defining ................................................................. 83
generated lines, controlling in list file ............................ 91
in-line routines ......................................................... 85
predefined symbol ..................................................... 87
processing .............................................................. 86
quote characters, specifying .......................................... 42
special characters, using .............................................. 84
assembler object file, specifying filename ....................... 44
assembly warning messages ........................................ 131
disabling ............................................................... 47
ASSIGN (assembler directive) ..................................... 78
assumptions (programming experience) ......................... 11

B
-B (assembler option) ................................................. 35
backtrace information See call frame information
backtrace information, defining ................................. 105
bitwise AND (assembler operator) .............................. 105
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

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 (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

AARM-9

Index
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. See assembler directives
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
END (assembler directive) ......................... 82
ENDR (assembler directive) ......................... 82
environment variables
   assembler ........................................... 20
   IASMARM ............................................ 20
   IASMARM_INC ....................................... 20
   :EOR: (assembler operator) ....................... 54
   EQ (CFI operator) .................................. 24
   EQU (assembler directive) ........................ 78
   equal (assembler operator) ...................... 131
   #error (assembler directive) .................... 95
   error messages
      format ............................................ 131
      maximum number, specifying .................... 37
      #error, using to display ....................... 97
   EVEN (assembler directive) ...................... 74
   EXITM (assembler directive) ..................... 74
   experience, programming .......................... 11
   expressions ......................................... 22
   extended command line file (extend.xcl) ........ 33, 38
   EXTERN (assembler directive) ..................... 70
   EXTWEAK (assembler directive) ................... 70

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

F
   -f (assembler option) ............................. 33, 38
   false value, in assembler expressions .......... 24
   fatal errors ....................................... 132
   __FILE__ (predefined symbol) .................... 26
   file extensions. See 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

IAR Assembler
Reference Guide for ARM

142
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 (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
HWRD (assembler operator) ............................... 59

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

-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
#ifdef (assembler directive) ......................... 95
#include files .......................................... 40
#include (assembler directive) .......................... 95
#include files, specifying ................................ 40
#include (assembler directive) ......................... 95
include files, disabling search for ..................... 39
include paths, specifying ................................ 40
INCLUDED (assembler directive) ......................... 102
installation directory ................................... 14
integer constants ........................................ 22
internal errors, assembler ................................ 132
invocation syntax ....................................... 19
in-line errors, assembler ................................ 132
in-line, using macros ................................... 87
italic coding, using macros ......................... 87
italic style, in this guide ................................ 14

-J (assembler option) ................................... 41

-L (assembler option) ................................... 41
-I (assembler option) ................................... 40
labels, See assembler labels ............................... 40
: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
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
:NOT: (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
LSTCNND (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 (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. See 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
MISRCA C, documentation ........................................... 13
:MOD: (assembler operator) ........................................ 56
MOD (CFI operator) .................................................. 115
  mode control directives ....................................... 72
module consistency ................................................ 69
module control directives ....................................... 69
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 (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 (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. See assembler operators
option summary .................................... 34
:OR: (assembler operator) ....................... 56
OR (CFI operator) .................................. 115
-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 (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

AARM-9
REPT (assembler directive) ............................................. 82
REPTC (assembler directive) ........................................... 82
REPTI (assembler directive) ............................................ 82
REQUIRE (assembler directive) .......................... 70
REQUIREB8 (assembler directive) ..................... 68
RSEG (assembler directive) ........................................ 74
RSHIFTA (CFI operator) .................................................. 115
RSHIPTL (CFI operator) ............................................... 116
RMODEL (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. See special function registers ....................... 60
: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
See also Include cross-reference
symbol control directives ....................................... 70
symbols
See also 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. See 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
ttrue 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
- **__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) .......................................... 36
- **-e** (assembler option) .......................................... 39
- **-f** (assembler option) .......................................... 39
- **-G** (assembler option) .......................................... 39
- **-g** (assembler option) .......................................... 39
- **-I** (assembler option) .......................................... 40
- **-i** (assembler option) .......................................... 40
- **-j** (assembler option) .......................................... 41
- **-L** (assembler option) .......................................... 41
- **-l** (assembler option) .......................................... 41
- **-M** (assembler option) .......................................... 42
- **-N** (assembler option) .......................................... 43
- **-n** (assembler option) .......................................... 43
- **-O** (assembler option) .......................................... 44
- **-p** (assembler option) .......................................... 44
- **-r** (assembler option) .......................................... 45
- **-S** (assembler option) .......................................... 45
Numerics

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