



**IAR Embedded
Workbench**

IAR Assembler Reference Guide

for Arm Limited's
Arm Cores

AARM-11

**IAR**
SYSTEMS

COPYRIGHT NOTICE

© 1999–2017 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, IAR Connect, C-SPY, C-RUN, C-STAT, IAR Visual State, visualSTATE, IAR KickStart Kit, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Arm, Cortex, Thumb, and TrustZone are registered trademarks of Arm Limited. EmbeddedICE is a trademark of Arm Limited. uC/OS-II and uC/OS-III are trademarks of Micrium, Inc. CMX-RTX is a trademark of CMX Systems, Inc. ThreadX is a trademark of Express Logic. RTX is a trademark of Quadros Systems. Fusion is a trademark of Unicoi Systems.

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

Eleventh edition: October 2017

Part number: AARM-11

This guide applies to version 8.2x of IAR Embedded Workbench® for Arm.

Internal reference: BB2, Mym8.0, asrct2010.3, V_110411, asrcarm7.80, IMAE.

Contents

Tables	11
Preface	13
Who should read this guide	13
How to use this guide	13
What this guide contains	14
Document conventions	14
Typographic conventions	14
Naming conventions	15
Introduction to the IAR Assembler for Arm	17
Introduction to assembler programming	17
Getting started	17
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	23
Symbols	24
Labels	24
Register symbols	25
Predefined symbols	25
Absolute and relocatable expressions	29
Expression restrictions	30

List file format	30
Header	30
Body	31
Summary	31
Symbol and cross-reference table	31
Programming hints	31
Accessing special function registers	31
Using C-style preprocessor directives	32
Tracking call frame usage	32
Call frame information overview	33
Call frame information in more detail	33
Defining a names block	34
Defining a common block	35
Annotating your source code within a data block	36
Specifying rules for tracking resources and the stack depth	37
Using CFI expressions for tracking complex cases	39
Stack usage analysis directives	39
Examples of using CFI directives	40
Assembler options	43
Using command line assembler options	43
Specifying options and their parameters	43
Extended command line file	44
Summary of assembler options	44
Description of assembler options	46
--arm	46
-B	46
-c	47
--cmse	47
--cpu	47
--cpu_mode	48
-D	48
-E	49
-e	49

--endian	49
-f	50
--fpu	50
-G	51
-g	51
-I	51
-i	52
-j	52
-L	53
-l	53
--legacy	54
-M	54
-N	55
--no_dwarf3_cfi	55
--no_it_verification	55
--no_literal_pool	55
--no_path_in_file_macros	56
-O	56
-o	57
-p	57
-r	58
-S	58
-s	58
--source_encoding	59
--suppress_vfe_header	59
--system_include_dir	60
-t	60
--thumb	60
-U	61
-w	61
-x	62

Assembler operators	63
Precedence of assembler operators	63
Summary of assembler operators	63
Parenthesis operator	63
Unary operators	64
Multiplicative arithmetic operators	64
Additive arithmetic operators	65
Shift operators	65
AND operators	65
OR operators	65
Comparison operators	66
Description of assembler operators	66
() Parenthesis	66
* Multiplication	66
+ Unary plus	67
+ Addition	67
– Unary minus	67
– Subtraction	68
/ Division	68
< Less than	68
<= Less than or equal	68
<>, != Not equal	69
=, == Equal	69
> Greater than	69
>= Greater than or equal	70
&& Logical AND	70
& Bitwise AND	70
~ Bitwise NOT	71
Bitwise OR	71
^ Bitwise exclusive OR	71
% Modulo	71
! Logical NOT	72
Logical OR	72

<< Logical shift left	72
>> Logical shift right	73
BYTE1 First byte	73
BYTE2 Second byte	73
BYTE3 Third byte	73
BYTE4 Fourth byte	74
DATE Current time/date	74
HIGH High byte	74
HWRD High word	75
LOW Low byte	75
LWRD Low word	75
SFB section begin	75
SFE section end	76
SIZEOF section size	76
UGT Unsigned greater than	77
ULT Unsigned less than	77
XOR Logical exclusive OR	78
Assembler directives	79
Summary of assembler directives	79
Description of assembler directives	83
Module control directives	83
Symbol control directives	86
Mode control directives	88
Section control directives	90
Value assignment directives	93
Conditional assembly directives	95
Macro processing directives	96
Listing control directives	105
C-style preprocessor directives	110
Data definition or allocation directives	115
Assembler control directives	117
Function directives	120
Call frame information directives for names blocks	121

Call frame information directives for common blocks	122
Call frame information directives for data blocks	123
Call frame information directives for tracking resources and CFAs	125
Call frame information directives for stack usage analysis	127
Assembler pseudo-instructions	129
Summary	129
Descriptions of pseudo-instructions	130
ADR (ARM)	130
ADR (CODE16)	131
ADR (THUMB)	131
ADRL (ARM)	132
ADRL (THUMB	133
LDR (ARM)	133
LDR (CODE16)	134
LDR (THUMB)	135
MOV (CODE16)	136
MOV32 (THUMB)	137
NOP (ARM)	137
NOP (CODE16)	137
Assembler diagnostics	139
Message format	139
Severity levels	139
Options for diagnostics	139
Assembler warning messages	139
Command line error messages	139
Assembler error messages	140
Assembler fatal error messages	140
Assembler internal error messages	140
Migrating to the IAR Assembler for Arm	141
Introduction	141
Thumb code labels	141

Alternative register names	142
Alternative mnemonics	143
Operator synonyms	144
Warning messages	145
The first register operand omitted	145
The first register operand duplicated	145
Immediate #0 omitted in Load/Store	145
Index	147

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	25
10: Symbol and cross-reference table	31
11: Code sample with backtrace rows and columns	40
12: Assembler options summary	44
13: Assembler directives summary	79
14: Module control directives	84
15: Symbol control directives	86
16: Mode control directives	88
17: Section control directives	91
18: Value assignment directives	94
19: Macro processing directives	97
20: Listing control directives	106
21: C-style preprocessor directives	111
22: Data definition or allocation directives	115
23: Assembler control directives	118
24: Call frame information directives names block	122
25: Call frame information directives common block	123
26: Call frame information directives for data blocks	124
27: Unary operators in CFI expressions	125
28: Binary operators in CFI expressions	125
29: Ternary operators in CFI expressions	126
30: Call frame information directives for tracking resources and CFAs	127
31: Call frame information directives for stack usage analysis	128

32: Pseudo-instructions	129
33: Alternative register names	142
34: Alternative mnemonics	143
35: Operator synonyms	144

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 for Arm. In addition, you should have working knowledge of the following:

- The architecture and instruction set of the Arm core (refer to the chip manufacturer's documentation)
- General assembler language programming
- Application development for embedded systems
- The operating system of your host computer.

How to use this guide

When you first begin using the IAR Assembler for Arm, you should read the chapter *Introduction to the IAR Assembler for Arm*.

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

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

What this guide contains

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

- *Introduction to the IAR Assembler for 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 pseudo-instructions that are accepted by the IAR Assembler for Arm.
- *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 migrating from an existing product to the IAR Assembler for Arm.

Document conventions

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

When referring to a directory in your product installation, for example `arm\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench N.n\arm\doc`, where the initial digit of the version number reflects the initial digit of the version number of the IAR Embedded Workbench shared components.

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:

Style	Used for
<code>computer</code>	<ul style="list-style-type: none"> • Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.

Table 1: Typographic conventions used in this guide





Style	Used for
<i>parameter</i>	A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file.
[option]	An optional part of a directive, where [and] are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
{option}	A mandatory part of a directive, where { and } are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
[option]	An optional part of a command.
[a b c]	An optional part of a command with alternatives.
{a b c}	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none"> • A cross-reference within this guide or to another guide. • Emphasis.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide (Continued)

NAMING CONVENTIONS

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

Brand name	Generic term
IAR Embedded Workbench® for Arm	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for Arm	the IDE
IAR C-SPY® Debugger for Arm	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for Arm	the compiler
IAR Assembler™ for Arm	the assembler

Table 2: Naming conventions used in this guide

Brand name	Generic term
IAR ILINK Linker™	ILINK, the linker
IAR DLIB Runtime Environment™	the DLIB runtime environment

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

Introduction to the IAR Assembler for Arm

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

Introduction to assembler programming

Even if you do not intend to write a complete application in assembler language, there might be situations where you find it necessary to write parts of the code in assembler, for example, when using mechanisms in 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 the Arm Limited 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:

- *Assembler invocation syntax*, page 19
- *Passing options*, page 20
- *Environment variables*, page 20
- *Error return codes*, page 20

You can use the assembler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *IAR Embedded Workbench® IDE User Guide for 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 50.

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

ENVIRONMENT VARIABLES

You can use these environment variables with the IAR Assembler:

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

Table 3: Assembler environment variables

For example, setting this environment variable always generates a list file with the name `temp.lst`:

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

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

ERROR RETURN CODES

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

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

Table 4: Assembler error return codes

Return code	Description
2	Errors occurred.

Table 4: Assembler error return codes (Continued)

Source format

The format of an assembler source line is as follows:

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

where the components are as follows:

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

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*.

These operands are valid in an expression:

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

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

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

INTEGER CONSTANTS

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

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

Commas and decimal points are not permitted.

The following types of number representation are supported:

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

Table 5: Integer constant formats

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

ASCII CHARACTER CONSTANTS

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

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

Table 6: ASCII character constant formats

FLOATING-POINT CONSTANTS

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

Floating-point numbers can be written in the format:

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

This table shows some valid examples:

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

Table 7: Floating-point constants

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

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

TRUE AND FALSE

In expressions a zero value is considered false, and a non-zero value is considered true.

Conditional expressions return the value 0 for false and 1 for true.

SYMBOLS

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

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

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

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

```
`strange#label`
```

Case is insignificant for built-in symbols like instructions, registers, operators, and directives. For user-defined symbols, case is by default significant but can be turned on and off using the **Case sensitive user symbols** (`-s`) assembler option. For more information, see `-s`, page 58.

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

Note that symbols and labels are byte addresses. See also *Data definition or allocation directives*, page 115.

LABELS

Symbols used for memory locations are referred to as labels.

Program location counter (PLC)

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

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

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


REGISTER SYMBOLS

This table shows the existing predefined register symbols:

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

Table 8: Predefined register symbols

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

PREDEFINED SYMBOLS

The IAR Assembler for Arm defines a set of symbols for use in assembler source files. The symbols provide information about the current assembly, allowing you to test them in preprocessor directives or include them in the assembled code.

These predefined symbols are available:

Symbol	Value
<code>__ARM_ADVANCED_SIMD__</code>	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor architecture has the Advanced SIMD architecture extension. The symbol is undefined for other cores.
<code>__ARM_ARCH</code>	This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_ARCH_ISA_ARM</code>	This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .

Table 9: Predefined symbols

Symbol	Value
<code>__ARM_ARCH_ISA_THUMB</code>	This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_ARCH_PROFILE</code>	This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_BIG_ENDIAN</code>	This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_FEATURE_CMSE</code>	An integer that is set based on the assembler option <code>--cpu</code> and <code>--cmse</code> . The symbol is set to 3 if the selected processor architecture has CMSE (Cortex-M security extensions) and the assembler option <code>--cmse</code> is specified. The symbol is set to 1 if the selected processor architecture has CMSE and the assembler option <code>--cmse</code> is not specified. The symbol is undefined for cores without CMSE.
<code>__ARM_FEATURE_CRC32</code>	This symbol is defined to 1 if the CRC32 instructions are supported (optional in Armv8-A/R). This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_FEATURE_CRYPTO</code>	This symbol is defined to 1 if the crypto instructions are supported (implies Armv8-A/R with Neon). This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_FEATURE_DIRECTED_ROUNDING</code>	This symbol is defined to 1 if the directed rounding and conversion instructions are supported. This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_FEATURE_DSP</code>	This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_FEATURE_FMA</code>	This symbol is defined to 1 if the FPU supports fused floating-point multiply-accumulate. This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_FEATURE_IDIV</code>	This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_FEATURE_NUMERIC_MAXMIN</code>	This symbol is defined to 1 if the floating-point numeric maximum and minimum instructions are supported. This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .

Table 9: Predefined symbols (Continued)

Symbol	Value
<code>__ARM_FP</code>	This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_MEDIA__</code>	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor architecture has the ARMv6 SIMD extension for multimedia. The symbol is undefined for other cores.
<code>__ARM_MPCORE__</code>	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor architecture has the Multiprocessing Extensions. The symbol is undefined for other cores.
<code>__ARM_NEON</code>	This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_NEON_FP</code>	This symbol is defined according to the <i>Arm C Language Extensions (ACLE)</i> .
<code>__ARM_PROFILE_M__</code>	An integer that is set based on the <code>--cpu</code> option. The symbol is set to 1 if the selected processor is a profile M core. The symbol is undefined for other cores.
<code>__ARMVFP__</code>	An integer that is set based on the <code>--fpu</code> option and that identifies whether floating-point instructions for a vector floating-point coprocessor have been enabled or not. The symbol is defined to <code>__ARMVFPV2__</code> , <code>__ARMVFPV3__</code> , or <code>__ARMVFPV4__</code> . These symbolic names can be used when testing the <code>__ARMVFP__</code> symbol. If floating-point instructions are disabled (default), the symbol is undefined.
<code>__ARMVFP_D16__</code>	An integer that is set based on the assembler option <code>--fpu</code> . The symbol is set to 1 if the selected FPU is a VFPv3 or VFPv4 unit with only 16 D registers. Otherwise, the symbol is undefined.
<code>__ARMVFP_FP16__</code>	An integer that is set based on the assembler option <code>--fpu</code> . The symbol is set to 1 if the selected FPU only supports 16-bit floating-point numbers. Otherwise, the symbol is undefined.
<code>__ARMVFP_SP__</code>	An integer that is set based on the assembler option <code>--fpu</code> . The symbol is set to 1 if the selected FPU only supports 32-bit single-precision. Otherwise, the symbol is undefined.

Table 9: Predefined symbols (Continued)

Symbol	Value
<code>__BUILD_NUMBER__</code>	A unique integer that identifies the build number of the assembler currently in use. The build number does not necessarily increase with an assembler that is released later.
<code>__CORE__</code>	An integer that identifies the chip core in use. The value reflects the setting of the assembler option <code>--cpu</code> . For information about the possible values, see the <i>IAR C/C++ Development Guide for Arm</i> .
<code>__DATE__</code>	The current date in <code>dd/Mmm/yyyy</code> format (string).
<code>__FILE__</code>	The name of the current source file (string).
<code>__IAR_SYSTEMS_ASM__</code>	IAR assembler identifier (number). Note that the number could be higher in a future version of the product. This symbol can be tested with <code>#ifdef</code> to detect whether the code was assembled by an assembler from IAR Systems.
<code>__IASMARM__</code>	An integer that is set to 1 when the code is assembled with the IAR Assembler for Arm.
<code>__LINE__</code>	The current source line number (number).
<code>__LITTLE_ENDIAN__</code>	Identifies the byte order in use. Expands to the number 1 when the code is compiled with the little-endian byte order, and to the number 0 when big-endian code is generated. Little-endian is the default.
<code>__TID__</code>	Target identity, consisting of two bytes (number). The high byte is the target identity, which is 0x4F (=decimal 79) for the IAR Assembler for Arm.
<code>__TIME__</code>	The current time in <code>hh:mm:ss</code> format (string).
<code>__VER__</code>	The version number in integer format; for example, version 6.21.2 is returned as 6021002 (number).

Table 9: Predefined symbols (Continued)

Including symbol values in code

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

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

```
name    timeOfAssembly
extern  printStr
section MYCODE:CODE(2)
```

```

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

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

```

Testing symbols for conditional assembly

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

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

```

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

```

For more information, see *Conditional assembly directives*, page 95.

ABSOLUTE AND RELOCATABLE EXPRESSIONS

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

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

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

For example, a program could define absolute and relocatable expressions as follows:

```

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

```

```

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

```

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

EXPRESSION RESTRICTIONS

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

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

No forward

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

No external

No external references in the expression are allowed.

Absolute

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

Fixed

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

List file format

The format of an assembler list file is as follows:

HEADER

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

BODY

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

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

SUMMARY

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

SYMBOL AND CROSS-REFERENCE TABLE

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

This information is provided for each symbol in the table:

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

Table 10: Symbol and cross-reference table

Programming hints

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

ACCESSING SPECIAL FUNCTION REGISTERS

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

Example

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

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

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

```
UA0_RBR DEFINE 0x40050000
```

USING C-STYLE PREPROCESSOR DIRECTIVES

The C-style preprocessor directives are processed before other assembler directives. Therefore, do not use preprocessor directives in macros and do not mix them with assembler-style comments. For more information about comments, see *Assembler control directives*, page 117.

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

Tracking call frame usage

In this section, these topics are described::

- *Call frame information overview*, page 33
- *Call frame information in more detail*, page 33

These tasks are described:

- *Defining a names block*, page 34
- *Defining a common block*, page 35
- *Annotating your source code within a data block*, page 36
- *Specifying rules for tracking resources and the stack depth*, page 37
- *Using CFI expressions for tracking complex cases*, page 39
- *Stack usage analysis directives*, page 39
- *Examples of using CFI directives*, page 40

For reference information, see:

- *Call frame information directives for names blocks*, page 121
- *Call frame information directives for common blocks*, page 122
- *Call frame information directives for data blocks*, page 123
- *Call frame information directives for tracking resources and CFAs*, page 125
- *Call frame information directives for stack usage analysis*, page 127

CALL FRAME INFORMATION OVERVIEW

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

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

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

CALL FRAME INFORMATION IN MORE DETAIL

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

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

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

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

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

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

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

DEFINING A NAMES BLOCK

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

Start and end a names block with the directives:

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

where *name* is the name of the block.

Only one names block can be open at a time.

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

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

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

The parameters are the name of the resource and the size of the resource in bits. The name must be one of the register names defined in the AEABI document *DWARF for the ARM architecture*. A virtual resource is a logical concept, in contrast to a “physical” resource such as a processor register. Virtual resources are usually used for the return address.

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

A resource can also be a composite resource, made up of at least two parts. To declare the composition of a composite resource, use the directive:

```
CFI RESOURCEPARTS resource part, part, ...
```

The parts are separated with commas. The resource and its parts must have been previously declared as resources, as described above.

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

```
CFI STACKFRAME cfa resource type
```

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

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

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

```
CFI BASEADDRESS cfa type
```

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

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

DEFINING A COMMON BLOCK

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

Start a common block with the directive:

```
CFI COMMON name USING namesblock
```

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

Declare the return address column with the directive:

```
CFI RETURNADDRESS resource type
```

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

Inside a common block, you can declare the initial value of a CFA or a resource by using the directives available for common blocks, see *Call frame information directives for common blocks*, page 122. For more information about how to use these directives, see *Specifying rules for tracking resources and the stack depth*, page 37 and *Using CFI expressions for tracking complex cases*, page 39.

End a common block with the directive:

```
CFI ENDCOMMON name
```

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

ANNOTATING YOUR SOURCE CODE WITHIN A DATA BLOCK

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

Start a data block with the directive:

```
CFI BLOCK name USING commonblock
```

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

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

```
CFI FUNCTION label
```

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

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

```
CFI NOFUNCTION
```

End a data block with the directive:

```
CFI ENDBLOCK name
```

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

Inside a data block, you can manipulate the values of the resources by using the directives available for data blocks, see *Call frame information directives for data blocks*, page 123. For more information on how to use these directives, see *Specifying rules for tracking resources and the stack depth*, page 37, and *Using CFI expressions for tracking complex cases*, page 39.

SPECIFYING RULES FOR TRACKING RESOURCES AND THE STACK DEPTH

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

- Rules for tracking resources

```
CFI resource { UNDEFINED | SAMEVALUE | CONCAT }
```

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

- Rules for tracking the stack depth (CFAs)

```
CFI cfa { NOTUSED | USED }
```

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

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

In those rare cases where the descriptive power of the simple rules are not enough, you can use a full *CFI expression* with dedicated *operators* to describe the information, see *Using CFI expressions for tracking complex cases*, page 39. However, whenever possible, you should always use a rule instead of a CFI expression.

Rules for tracking resources

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

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

```
CFI R11 SAMEVALUE
```

To declare that a resource is not tracked, use `UNDEFINED` as location. Conceptually, this declares that the resource does not have to be restored (when going back one call frame) because it is not tracked. Usually it is only meaningful to use it to declare the initial location of a resource. For example, to declare that `R11` is a scratch register and does not have to be restored, use the directive:

```
CFI R11 UNDEFINED
```

To declare that a resource is temporarily stored in another resource, use the resource name as its location. For example, to declare that a register R11 is temporarily located in a register R12 (and should be restored from that register), use the directive:

```
CFI R11 R12
```

To declare that a resource is currently located somewhere on the stack, use `FRAME(cfa, offset)` as location for the resource, where *cfa* is the CFA identifier to use as “frame pointer” and *offset* is an offset relative the CFA. For example, to declare that a register R11 is located at offset `-4` counting from the frame pointer `CFA_SP`, use the directive:

```
CFI R11 FRAME(CFA_SP, -4)
```

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

```
CFI RET CONCAT
```

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

Rules for tracking the stack depth (CFAs)

In contrast to the rules for resources, the rules for CFAs describe the address of the beginning of the call frame. The call frame often includes the return address pushed by the assembler call instruction. The CFA rules describe how to compute the address of the beginning of the current stack frame.

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

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

```
CFI CFA_SP NOTUSED
```

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

```
CFI CFA_SP SP + 4
```

USING CFI EXPRESSIONS FOR TRACKING COMPLEX CASES

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

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

- Unary operators
- Binary operators
- Ternary operators

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

In this example, R12 is restored to its original value. However, instead of saving it, the effect of the two post increments is undone by the subtract instruction.

AddTwo:

```

cfi block addTwoBlock using myCommon
cfi function addTwo
cfi nocalis
cfi r12 samevalue
add @r12+, r13
cfi r12 sub(r12, 2)
add @r12+, r13
cfi r12 sub(r12, 4)
sub #4, r12
cfi r12 samevalue
ret
cfi endblock addTwoBlock

```

For more information about the syntax for using the operators in CFI expressions, see *Call frame information directives for tracking resources and CFAs*, page 125.

STACK USAGE ANALYSIS DIRECTIVES

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

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

EXAMPLES OF USING CFI DIRECTIVES

The following is 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 call frame information. 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, in other words, the value of the stack pointer after returning from the function.

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

Table 11: Code sample with backtrace rows and columns

Each row describes the state of the tracked resources *before* the execution of the instruction. As an example, for the `MOV R1, R0` instruction, the original value of the R1 register is located in the R0 register and the top of the function frame (the CFA column) is `R13 + 0`. The row at address `0000` is the initial row and the result of the calling convention used for the function.

The R14 column is the return address column—in other words, the location of the return address. 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. Some of the registers are undefined because they do not need to be restored on exit from the function.

Defining the names block

The names block for the small example above would be:

```

cfi      names ArmCore
cfi      stackframe cfa r13 DATA
cfi      resource r0:32, r1:32, r2:32, r3:32
cfi      resource r4:32, r5:32, r6:32, r7:32
cfi      resource r8:32, r9:32, r10:32, r11:32
cfi      resource r12:32, r13:32, r14:32
cfi      endnames ArmCore

```


Defining the common block

```

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

```

Note: R13 cannot be changed using a CFI directive because it is the resource associated with CFA.

Defining the data block

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.

```

                section MYCODE:CODE(2)

                cfi      block trivialBlock using trivialCommon
                cfi      function func1

                thumb

func1           push     {r1,lr}

                cfi      r1  frame(cfa, -8)
                cfi      r14 frame(cfa, -4)
                cfi      cfa r13+8

                movs     r1,#4

                cfi      funcall1 func2

                bl      func2
                pop     {r0,lr}

                cfi      r1  r0
                cfi      r14 samevalue
                cfi      cfa r13

                mov     r1,r0

```

```
cfi    r1 samevalue
bx     lr
cfi    endblock trivialBlock
end
```

Assembler options

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

Using command line assembler options

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



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

SPECIFYING OPTIONS AND THEIR PARAMETERS

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 (that may be prefixed by a path), included after the option letter with a separating space. For example, to generate a list file with the name `list.lst`:

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

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

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

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

EXTENDED COMMAND LINE FILE

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

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

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

Summary of assembler options

This table summarizes the assembler options available from the command line:

Command line option	Description
<code>--arm</code>	Sets the default mode for the assembler directive <code>CODE</code> to Arm.
<code>-B</code>	Macro execution information
<code>-c</code>	Conditional list
<code>--cmse</code>	Enables CMSE secure object generation
<code>--cpu</code>	Core configuration
<code>--cpu_mode</code>	Sets the mode for the assembler directive <code>CODE</code> .
<code>-D</code>	Defines preprocessor symbols
<code>-E</code>	Maximum number of errors
<code>-e</code>	Generates code in big-endian byte order
<code>--endian</code>	Specifies the byte order for code and data
<code>-f</code>	Extends the command line
<code>--fpu</code>	Floating-point coprocessor architecture configuration
<code>-G</code>	Opens standard input as source
<code>-g</code>	Disables the automatic search for system include files
<code>-I</code>	Adds a search path for a header file
<code>-i</code>	Lists <code>#included</code> text

Table 12: Assembler options summary

Command line option	Description
-j	Enables alternative register names, mnemonics, and operators
-L	Generates a list file to path
-l	Generates a list file
--legacy	Generates code linkable with older toolchains
-M	Macro quote characters
-N	Omits header from the assembler listing
--no_dwarf3_cfi	Suppresses generation of DWARF 3 Call Frame Information instructions
--no_it_verification	Suppresses the verification of the condition of instructions following an IT instruction
--no_path_in_file_macros	Removes the path from the return value of the symbols <code>__FILE__</code> and <code>__BASE_FILE__</code>
-O	Sets the object filename to path
-o	Sets the object filename
-p	Sets the number of lines per page in the list file
-r	Generates debug information.
-S	Sets silent operation
-s	Case-sensitive user symbols
--source_encoding	Specifies the encoding for source files
--suppress_vfe_header	Suppresses the generation of VFE header information
--system_include_dir	Specifies the path for system include files
-t	Tab spacing
--thumb	Sets the default mode for the assembler directive <code>CODE</code> to Thumb
-U	Undefines a symbol
--use_unix_directory_separators	Uses <code>/</code> as directory separator in paths
-w	Disables warnings
-x	Includes cross-references

Table 12: Assembler options summary (Continued)

Description of assembler options

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



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

--arm

Syntax	--arm
Description	Use this option to make Arm the default mode for the assembler directive <code>CODE</code> .
See also	--cpu_mode, page 48



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

-B

Syntax	-B
Description	Use this option to make the assembler print macro execution information to the standard output stream for every call to a macro. The information consists of: <ul style="list-style-type: none"> • The name of the macro • The definition of the macro • The arguments to the macro • The expanded text of the macro. <p>This option is mainly used in conjunction with the list file options <code>-L</code> or <code>-l</code>.</p>
See also	<code>-L</code> , page 53.



Project>Options>Assembler >List>Macro execution info

-c

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

Parameters

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

Description

Use this option to control the contents of the assembler list file.
This option is mainly used in conjunction with the list file options `-L` or `-l`.

See also

`-L`, page 53.



To set related options, select:

Project>Options>Assembler>List

--cmse

Syntax `--cmse`

Description

Use this option to target secure mode in TrustZone for ARMv8-M. This option enables access to system registers with the suffix `_NS` using the instructions `MRS` and `MSR`, and enables the use of the instructions `SG`, `TTA`, `TTAT`, `BLXNS`, and `BXNS`.



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

--cpu

Syntax `--cpu target_core`

Parameters

`target_core` Can be values such as `ARM7TDMI` or architecture versions, for example `4T`. The default value is `ARM7TDMI`.

Description

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

See also

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



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

--cpu_mode

Syntax

```
--cpu_mode {arm|thumb}
```

Parameters

arm (default)	Uses Arm mode.
thumb	Uses Thumb mode.

Description

Use this option to select the mode for the assembler directive `CODE`.



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

-D

Syntax

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

Parameters

<i>symbol</i>	The name of the symbol you want to define.
<i>value</i>	The value of the symbol. If no value is specified, 1 is used.

Description

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

Example

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

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

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

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


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

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



Project>Options>Assembler>Preprocessor>Defined symbols

-E

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



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

-e

Syntax	<code>-e</code>	
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	<code>--endian {<i>little</i> <i>l</i> <i>big</i> <i>b</i>}</code>	
Parameters	<i>little</i> , <i>l</i> (default)	Specifies little-endian byte order.
	<i>big</i> , <i>b</i>	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 43.

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
```

See also *Extended command line file*, page 44.



To set this option, use:

Project>Options>Assembler>Extra Options

--fpu

Syntax `--fpu fpu_variant`

Parameters

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

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

See also [The IAR C/C++ Development Guide for Arm](#) for a complete list of coprocessor architecture variants.



Project>Options>General Options>Target>FPU

-G

Syntax `-G`

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

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



This option is not available in the IDE.

-g

Syntax `-g`

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



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

-I

Syntax `-Ipath`

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 144 and the chapter *Migrating to the IAR Assembler for Arm*.




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


-L

Syntax	<code>-L[path]</code>	
Parameters	No parameter	Generates a listing with the same name as the source file, but with the filename extension <code>lst</code> .
	<code>path</code>	The path to the destination of the list file. Note that you must not include a space before the path.
Description	By default, the assembler does not generate a list file. Use this option to make the assembler generate one and send it to the file <code>[path]sourcename.lst</code> . <code>-L</code> cannot be used at the same time as <code>-l</code> .	
Example	To send the list file to <code>list\prog.lst</code> rather than the default <code>prog.lst</code> : <code>iasmarm prog -Llist\</code>	
		To set related options, select: Project>Options>Assembler >List


-l

Syntax	<code>-l filename</code>	
Parameters	<code>filename</code>	The output is stored in the specified file. Note that you must include a space before the filename. If no extension is specified, <code>lst</code> is used.
	For information about specifying a filename, see <i>Using command line assembler options</i> , page 43.	
Description	Use this option to make the assembler generate a listing and send it to the file <code>filename</code> . By default, the assembler does not generate a list file.	
	To generate a list file with the default filename, use the <code>-L</code> option instead.	
		To set related options, select: Project>Options>Assembler >List

--legacy

Syntax	<code>--legacy {RVCT3.0}</code>	
Parameters	<code>RVCT3.0</code>	Specifies the linker in RVCT3.0. Use this parameter together with the <code>--aeabi</code> 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	<code>-Mab</code>	
Parameters	<code>ab</code>	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 <i>a</i> and <i>b</i> respectively. By default, the characters are <code><</code> and <code>></code> . The <code>-M</code> option allows you to change the quote characters to suit an alternative convention or simply to allow a macro argument to contain <code><</code> or <code>></code> themselves.	
Example	For example, using the option: <code>-M[]</code> in the source you would write, for example: <code>print [>]</code> to call a macro <code>print</code> with <code>></code> as the argument. Note: Depending on your host environment, it might be necessary to use quote marks with the macro quote characters, for example: <code>iasmarm filename -M'<>'</code>	 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 <code>-L</code> or <code>-l</code> .
See also	<code>-L</code> , page 53.



Project>Options>Assembler >List>Include header

--no_dwarf3_cfi

Syntax	<code>--no_dwarf3_cfi</code>
Description	Use this option to suppress generation of DWARF 3 call frame information instructions. This can lead to a degraded debugging experience, but might allow loading in a debugger that does not support DWARF 3.



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

--no_it_verification

Syntax	<code>--no_it_verification</code>
Description	Use this option to suppress the verification of the condition of instructions following an <code>IT</code> instruction.



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

--no_literal_pool

Syntax	<code>--no_literal_pool</code>
Description	Use this option for code that should run from a memory address range where read access via the data bus is prohibited.

With the option `--no_literal_pool`, the assembler uses the `MOV32` pseudo-instruction instead of using a literal pool for `LDR`. Note that other instructions can still cause read access via the data bus.

The option also affects the automatic library selection performed by the linker. An IAR-specific ELF attribute is used for determining whether libraries compiled with the option `--no_literal_pool` should be used.

The option `--no_literal_pool` is only allowed for cores with the architectures ARMv6-M, ARMv7-M, and ARMv8-M.

See also

The compiler and linker options with the same name in the *IAR C/C++ Development Guide for Arm*.



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

--no_path_in_file_macros

Syntax `--no_path_in_file_macros`

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



This option is not available in the IDE.

-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

<i>filename</i>	The object code is stored in the specified file.
<i>directory</i>	The object code is stored in a file (filename extension <code>.o</code>) which is stored in the specified directory.

For information about specifying a filename or directory, see *Using command line assembler options*, page 43.

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

<i>lines</i>	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 53
`-l`, page 53.



Project>Options>Assembler>List>Lines/page

-r

Syntax `-r`

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



Project>Options>Assembler>Output>Generate debug information

-S

Syntax `-S`

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

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



This option is not available in the IDE.

-s

Syntax `-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

--source_encoding

Syntax

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

Parameters

`locale` The default source encoding is the system locale encoding.

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

Description

When reading a source file with no Byte Order Mark (BOM), use this option to specify the encoding.

If this option is not specified and the source file does not have a BOM, the Raw encoding will be used.

See also

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



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

--suppress_vfe_header

Syntax

```
--suppress_vfe_header
```

Description

Use this option to suppress the automatic generation of VFE (Virtual Function Elimination) header information in generated object code.

This option is useful in two cases:

- Making sure that the linker VFE optimization is not automatically turned on.
- Manually supplying VFE information in the assembler source code.


See also

The linker option `--vfe` in the *IAR C/C++ Development Guide for Arm*.



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

--system_include_dir

Syntax	<code>--system_include_dir path</code>
Parameters	<i>path</i> The path to the system include files.
Description	By default, the assembler automatically locates the system include files. Use this option to explicitly specify a different path to the system include files. This might be useful if you have not installed IAR Embedded Workbench in the default location.
	 This option is not available in the IDE.

-t

Syntax	<code>-tn</code>
Parameters	<i>n</i> The tab spacing; must be in the range 2 to 9.
Description	By default, the assembler sets 8 character positions per tab stop. Use this option to specify a different tab spacing. This option is useful in conjunction with the list options <code>-L</code> or <code>-l</code> .
See also	<code>-L</code> , page 53 <code>-l</code> , page 53.

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

Syntax	<code>--thumb</code>
Description	Use this option to make Thumb the default mode for the assembler directive <code>CODE</code> .
See also	<code>--cpu_mode</code> , page 48

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

-U

Syntax	<code>-U<i>symbol</i></code>
Parameters	<i>symbol</i> 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 <code>-D</code> option or source definition.
Example	To use the name of the predefined symbol <code>__TIME__</code> for your own purposes, you could undefine it with: <code>iasmarm prog -U__TIME__</code>
See also	<i>Predefined symbols</i> , page 25.



This option is not available in the IDE.

-w

Syntax	<code>-w[+ - +n -n +m-n -m-n] [s]</code>																
Parameters	<table> <tr> <td>No parameter</td> <td>Disables all warnings.</td> </tr> <tr> <td>+</td> <td>Enables all warnings.</td> </tr> <tr> <td>-</td> <td>Disables all warnings.</td> </tr> <tr> <td>+n</td> <td>Enables just warning <i>n</i>.</td> </tr> <tr> <td>-n</td> <td>Disables just warning <i>n</i>.</td> </tr> <tr> <td>+m-n</td> <td>Enables warnings <i>m</i> to <i>n</i>.</td> </tr> <tr> <td>-m-n</td> <td>Disables warnings <i>m</i> to <i>n</i>.</td> </tr> <tr> <td>s</td> <td>Generates the exit code 1 if a warning message is produced. By default, warnings generate exit code 0.</td> </tr> </table>	No parameter	Disables all warnings.	+	Enables all warnings.	-	Disables all warnings.	+n	Enables just warning <i>n</i> .	-n	Disables just warning <i>n</i> .	+m-n	Enables warnings <i>m</i> to <i>n</i> .	-m-n	Disables warnings <i>m</i> to <i>n</i> .	s	Generates the exit code 1 if a warning message is produced. By default, warnings generate exit code 0.
No parameter	Disables all warnings.																
+	Enables all warnings.																
-	Disables all warnings.																
+n	Enables just warning <i>n</i> .																
-n	Disables just warning <i>n</i> .																
+m-n	Enables warnings <i>m</i> to <i>n</i> .																
-m-n	Disables warnings <i>m</i> to <i>n</i> .																
s	Generates the exit code 1 if a warning message is produced. By default, warnings generate exit code 0.																
Description	By default, the assembler displays a warning message when it detects an element of the source code which is legal in a syntactical sense, but might contain a programming error.																

Use this option to disable all warnings, a single warning, or a range of warnings.

Note that the `-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 139.

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 53

`-l`, page 53.



Project>Options>Assembler>List>Include cross reference

Assembler operators

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

Precedence of assembler operators

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

Note: The operator synonyms are enabled by the option `-j`. See also the chapter *Migrating to the LAR Assembler for Arm*.

PARENTHESIS OPERATOR

Precedence: 1

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

UNARY OPERATORS

Precedence: 1

+	Unary plus.
-	Unary minus.
!, :LNOT:	Logical NOT.
~, :NOT:	Bitwise NOT.
LOW	Low byte.
HIGH	High byte.
BYTE1	First byte.
BYTE2	Second byte.
BYTE3	Third byte.
BYTE4	Fourth byte.
LWRD	Low word.
HWRD	High word.
DATE	Current time/date.
SFB	Section begin.
SFE	Section end.
SIZEOF	Section size.

MULTIPLICATIVE ARITHMETIC OPERATORS

Precedence: 2

*	Multiplication.
/	Division.
%, :MOD:	Modulo.

ADDITIVE ARITHMETIC OPERATORS

Precedence: 3

+	Addition.
-	Subtraction.

SHIFT OPERATORS

Precedence: 2.5-4

>>	Logical shift right (4).
:SHR:	Logical shift right (2.5).
<<	Logical shift left (4).
:SHL:	Logical shift left (2.5).

AND OPERATORS

Precedence: 3-8

&&	Logical AND (5).
:LAND:	Logical AND (8).
&	Bitwise AND (5).
:AND:	Bitwise AND (3).

OR OPERATORS

Precedence: 3-8

, :LOR:	Logical OR (6).
	Bitwise OR (6).
:OR:	Bitwise OR (3).
XOR	Logical exclusive OR (6).
:LEOR:	Logical exclusive OR (8).
^	Bitwise exclusive OR (6).

:EOR: Bitwise exclusive OR (3).

COMPARISON OPERATORS

Precedence: 7

=, ==	Equal.
<>, !=	Not equal.
>	Greater than.
<	Less than.
UGT	Unsigned greater than.
ULT	Unsigned less than.
>=	Greater than or equal.
<=	Less than or equal.

Description of assembler operators

This section gives detailed descriptions of each assembler operator.

See also *Expressions, operands, and operators*, page 22.

() Parenthesis

Precedence	1
Description	(and) group expressions to be evaluated separately, overriding the default precedence order.
Example	1+2*3 -> 7 (1+2)*3 -> 9

* Multiplication

Precedence	2
Description	* produces the product of its two operands. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.

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

+ Unary plus

Precedence 1

Description Unary plus operator; performs nothing.

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

+ Addition

Precedence 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

Precedence 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

Precedence	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	<pre>92-19 -> 73 -2-2 -> -4 -2--2 -> 0</pre>

/ Division

Precedence	2
Description	/ produces the integer quotient of the left operand divided by the right operator. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer.
Example	<pre>9/2 -> 4 -12/3 -> -4 9/2*6 -> 24</pre>

< Less than

Precedence	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	<pre>-1 < 2 -> 1 2 < 1 -> 0 2 < 2 -> 0</pre>

<= Less than or equal

Precedence	7
Description	<= evaluates to 1 (true) if the left operand has a numeric value that is lower than or equal to the right operand, otherwise it is 0 (false).

Example

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

<>, != Not equal

Precedence 7

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

Example

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

=, == Equal

Precedence 7

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

Example

```
1 = 2 -> 0
2 == 2 -> 1
'ABC' = 'ABCD' -> 0
```

> Greater than

Precedence 7

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

Example

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

>= Greater than or equal

Precedence	7
Description	>= evaluates to 1 (true) if the left operand is equal to or has a higher numeric value than the right operand, otherwise it is 0 (false).
Example	<pre>1 >= 2 -> 0 2 >= 1 -> 1 1 >= 1 -> 1</pre>

&& Logical AND

Precedence	5 The precedence of :LAND: is 8.
Description	&& or the synonym :LAND: performs logical AND between its two integer operands. If both operands are non-zero the result is 1 (true), otherwise it is 0 (false).
Example	<pre>1010B && 0011B -> 1 1010B && 0101B -> 1 1010B && 0000B -> 0</pre>

& Bitwise AND

Precedence	5 The precedence of :AND: is 3.
Description	& or the synonym :AND: performs bitwise AND between the integer operands. Each bit in the 32-bit result is the logical AND of the corresponding bits in the operands.
Example	<pre>1010B & 0011B -> 0010B 1010B & 0101B -> 0000B 1010B & 0000B -> 0000B</pre>

~ Bitwise NOT

Precedence	1
Description	~ or the synonym :NOT: performs bitwise NOT on its operand. Each bit in the 32-bit result is the complement of the corresponding bit in the operand.
Example	~ 1010B -> 1111111111111111111111111111111110101B

| Bitwise OR

Precedence	6 The precedence of :OR: is 3.
Description	or the synonym :OR: performs bitwise OR on its operands. Each bit in the 32-bit result is the inclusive OR of the corresponding bits in the operands.
Example	1010B 0101B -> 1111B 1010B 0000B -> 1010B

^ Bitwise exclusive OR

Precedence	6 The precedence of :EOR: is 3.
Description	^ or the synonym :EOR: performs bitwise XOR on its operands. Each bit in the 32-bit result is the exclusive OR of the corresponding bits in the operands.
Example	1010B ^ 0101B -> 1111B 1010B ^ 0011B -> 1001B

% Modulo

Precedence	2
Description	% or the synonym :MOD: produces the remainder from the integer division of the left operand by the right operand. The operands are taken as signed 32-bit integers and the result is also a signed 32-bit integer. X % Y is equivalent to X-Y*(X/Y) using integer division.

Example

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

! Logical NOT

Precedence 1

Description ! or the synonym :LNOT: negates a logical argument.

Example

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

|| Logical OR

Precedence 6

Description || or the synonym :LOR: performs a logical OR between two integer operands.

Example

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

<< Logical shift left

Precedence 4

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

Note: The precedence of :SHL: is 2.5.

Example

```
00011100B << 3 -> 11100000B
00000111111111111111B << 5 -> 11111111111100000B
14 << 1 -> 28
```


>> Logical shift right

Precedence	4
Description	>> or the synonym :SHR: shifts the left operand, which is always treated as <code>unsigned</code> , 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

Precedence	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	BYTE1 0xABCD -> 0xCD

BYTE2 Second byte

Precedence	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	BYTE2 0x12345678 -> 0x56

BYTE3 Third byte

Precedence	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	BYTE3 0x12345678 -> 0x34

BYTE4 Fourth byte

Precedence	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	BYTE4 0x12345678 -> 0x12

DATE Current time/date

Precedence	1
Description	DATE gets the time when the current assembly began. The DATE operator takes an absolute argument (expression) and returns: DATE 1 Current second (0–59). DATE 2 Current minute (0–59). DATE 3 Current hour (0–23). DATE 4 Current day (1–31). DATE 5 Current month (1–12). DATE 6 Current year MOD 100 (1998 ->98, 2000 ->00, 2002 ->02).
Example	To specify the date of assembly: today: DC8 DATE 5, DATE 4, DATE 3

HIGH High byte

Precedence	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

Precedence	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

Precedence	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

Precedence	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

Syntax	<code>SFB(<i>section</i> [{+ -}<i>offset</i>])</code>	
Precedence	1	
Parameters	<i>section</i>	The name of a section, which must be defined before SFB is used.
	<i>offset</i>	An optional offset from the start address. The parentheses are optional if <i>offset</i> is omitted.
Description	SFB accepts a single operand to its right. The operator evaluates to the absolute address of the first byte of that section. This evaluation occurs at linking time.	

Example

```

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

```

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

SFE section end**Syntax**

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

Precedence

1

Parameters

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

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

Description

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

Example

```

name      sectionEnd
section MYCODE:CODE(2) ; Forward declaration
                        ; of MYCODE.
section MYCONST:CONST(2)
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**Syntax**

`SIZEOF section`

Precedence

1

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

UGT Unsigned greater than

Precedence	7
Description	<code>UGT</code> evaluates to 1 (true) if the left operand has a larger value than the right operand, otherwise it is 0 (false). The operation treats the operands as unsigned values.
Example	<pre> 2 UGT 1 -> 1 -1 UGT 1 -> 1 </pre>

ULT Unsigned less than

Precedence	7
Description	<code>ULT</code> evaluates to 1 (true) if the left operand has a smaller value than the right operand, otherwise it is 0 (false). The operation treats the operands as unsigned values.
Example	<pre> 1 ULT 2 -> 1 -1 ULT 2 -> 0 </pre>

XOR Logical exclusive OR

Precedence 6

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

Note: The precedence of :LEOR: is 8.

Example 0101B XOR 1010B -> 0
0101B XOR 0000B -> 1

Assembler directives

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

Summary of assembler directives

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

- *Module control directives*, page 83
- *Symbol control directives*, page 86
- *Mode control directives*, page 88
- *Section control directives*, page 90
- *Value assignment directives*, page 93
- *Conditional assembly directives*, page 95
- *Macro processing directives*, page 96
- *Listing control directives*, page 105
- *C-style preprocessor directives*, page 110
- *Data definition or allocation directives*, page 115
- *Assembler control directives*, page 117
- *Function directives*, page 120
- *Call frame information directives for names blocks*, page 121.
- *Call frame information directives for common blocks*, page 122
- *Call frame information directives for data blocks*, page 123
- *Call frame information directives for tracking resources and CFAs*, page 125
- *Call frame information directives for stack usage analysis*, page 127

This table gives a summary of all the assembler directives:

Directive	Description	Section
<code>_args</code>	Is set to number of arguments passed to macro.	Macro processing
<code>\$</code>	Includes a file.	Assembler control
<code>#define</code>	Assigns a value to a label.	C-style preprocessor
<code>#elif</code>	Introduces a new condition in an <code>#if...#endif</code> block.	C-style preprocessor

Table 13: Assembler directives summary

Directive	Description	Section
<code>#else</code>	Assembles instructions if a condition is false.	C-style preprocessor
<code>#endif</code>	Ends an <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> block.	C-style preprocessor
<code>#error</code>	Generates an error.	C-style preprocessor
<code>#if</code>	Assembles instructions if a condition is true.	C-style preprocessor
<code>#ifdef</code>	Assembles instructions if a symbol is defined.	C-style preprocessor
<code>#ifndef</code>	Assembles instructions if a symbol is undefined.	C-style preprocessor
<code>#include</code>	Includes a file.	C-style preprocessor
<code>#message</code>	Generates a message on standard output.	C-style preprocessor
<code>#pragma</code>	Recognized but ignored.	C-style preprocessor
<code>#undef</code>	Undefines a label.	C-style preprocessor
<code>/*comment*/</code>	C-style comment delimiter.	Assembler control
<code>//</code>	C++ style comment delimiter.	Assembler control
<code>=</code>	Assigns a permanent value local to a module.	Value assignment
<code>AAPCS</code>	Sets module attributes.	Module control
<code>ALIAS</code>	Assigns a permanent value local to a module.	Value assignment
<code>ALIGN</code>	Aligns the program location counter by inserting zero-filled bytes.	Section control
<code>ALIGNRAM</code>	Aligns the program location counter.	Section control
<code>ALIGNROM</code>	Aligns the program location counter by inserting zero-filled bytes.	Section control
<code>ARM</code>	Interprets subsequent instructions as 32-bit (Arm) instructions.	Mode control
<code>ASSIGN</code>	Assigns a temporary value.	Value assignment
<code>CASEOFF</code>	Disables case sensitivity.	Assembler control
<code>CASEON</code>	Enables case sensitivity.	Assembler control
<code>CFI</code>	Specifies call frame information.	Call frame information
<code>CODE</code>	Interprets subsequent instructions as Arm or Thumb instructions, depending on the setting of related assembler options.	Mode control
<code>CODE16</code>	Interprets subsequent instructions as 16-bit (Thumb) instructions. Replaced by <code>THUMB</code> .	Mode control
<code>CODE32</code>	Interprets subsequent instructions as 32-bit (Arm) instructions. Replaced by <code>ARM</code> .	Mode control

Table 13: Assembler directives summary (Continued)

Directive	Description	Section
COL	Sets the number of columns per page. Retained for backward compatibility reasons; recognized but ignored.	Listing control
DATA	Defines an area of data within a code section.	Mode control
DC8	Generates 8-bit constants, including strings.	Data definition or allocation
DC16	Generates 16-bit constants.	Data definition or allocation
DC24	Generates 24-bit constants.	Data definition or allocation
DC32	Generates 32-bit constants.	Data definition or allocation
DCB	Generates 8-bit byte constants, including strings.	Data definition or allocation
DCD	Generates 32-bit long word constants.	Data definition or allocation
DCW	Generates 16-bit word constants, including strings.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DS8	Allocates space for 8-bit integers.	Data definition or allocation
DS16	Allocates space for 16-bit integers.	Data definition or allocation
DS24	Allocates space for 24-bit integers.	Data definition or allocation
DS32	Allocates space for 32-bit integers.	Data definition or allocation
ELSE	Assembles instructions if a condition is false.	Conditional assembly
ELSEIF	Specifies a new condition in an IF...ENDIF block.	Conditional assembly
END	Ends the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing

Table 13: Assembler directives summary (Continued)

Directive	Description	Section
ENDR	Ends a repeat structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
EVEN	Aligns the program counter to an even address.	Section control
EXITM	Exits prematurely from a macro.	Macro processing
EXTERN	Imports an external symbol.	Symbol control
EXTWEAK	Imports an external symbol (which can be undefined).	Symbol control
IF	Assembles instructions if a condition is true.	Conditional assembly
IMPORT	Imports an external symbol.	Symbol control
INCLUDE	Includes a file.	Assembler control
LIBRARY	Begins a module; an alias for PROGRAM and NAME.	Module control
LOCAL	Creates symbols local to a macro.	Macro processing
LSTCND	Controls conditional assembler listing.	Listing control
LSTCOD	Controls multi-line code listing.	Listing control
LSTEXP	Controls the listing of macro generated lines.	Listing control
LSTMAC	Controls the listing of macro definitions.	Listing control
LSTOUT	Controls assembler-listing output.	Listing control
LSTPAG	Retained for backward compatibility reasons. Recognized but ignored.	Listing control
LSTREP	Controls the listing of lines generated by repeat directives.	Listing control
LSTXRF	Generates a cross-reference table.	Listing control
LTORG	Directs the current literal pool to be assembled immediately following the directive.	Assembler control
MACRO	Defines a macro.	Macro processing
MODULE	Begins a module; an alias for PROGRAM and NAME.	Module control
NAME	Begins a program module.	Module control
ODD	Aligns the program location counter to an odd address.	Section control
OVERLAY	Recognized but ignored.	Symbol control
PAGE	Retained for backward compatibility reasons.	Listing control
PAGSIZ	Retained for backward compatibility reasons.	Listing control

Table 13: Assembler directives summary (Continued)

Directive	Description	Section
PRESERVE8	Sets a module attribute.	Module control
PROGRAM	Begins a module.	Module control
PUBLIC	Exports symbols to other modules.	Symbol control
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.	Symbol control
RADIX	Sets the default base.	Assembler control
REPT	Assembles instructions a specified number of times.	Macro processing
REPTC	Repeats and substitutes characters.	Macro processing
REPTI	Repeats and substitutes strings.	Macro processing
REQUIRE	Forces a symbol to be referenced.	Symbol control
REQUIRE8	Sets a module attribute.	Module control
RSEG	Begins a section.	Section control
RTMODEL	Declares runtime model attributes.	Module control
SECTION	Begins a section.	Section control
SECTION_TYPE	Sets ELF type and flags for a section.	Section control
SETA	Assigns a temporary value.	Value assignment
THUMB	Interprets subsequent instructions as Thumb execution-mode instructions.	Mode control

Table 13: Assembler directives summary (Continued)

Description of assembler directives

The following pages give reference information about the assembler directives.

Module control directives

Syntax	AAPCS [<i>modifier</i> [...]]
	END
	NAME <i>symbol</i>
	PRESERVE8
	PROGRAM <i>symbol</i>
	REQUIRE8

```
RTMODEL key, value
```

Parameters

<i>key</i>	A text string specifying the key.
<i>modifier</i>	An AAPCS extension; possible values are INTERWORK, VFP, VFP_COMPATIBLE, ROPI, RWPI, RWPI_COMPATIBLE. Modifiers can be combined to specify AAPCS variants.
<i>symbol</i>	Name assigned to module.
<i>value</i>	A text string specifying the value.

Description

Module control directives are used for marking the beginning and end of source program modules, and for assigning names to them. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 30.

Directive	Description	Expression restrictions
END	Ends the assembly of the last module in a file.	Locally defined symbols plus offset or integer constants
NAME	Begins a module; alias to PROGRAM.	No external references Absolute
PROGRAM	Begins a module.	No external references Absolute
RTMODEL	Declares runtime model attributes.	Not applicable

Table 14: Module control directives

Beginning a program module

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

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

Beginning a module

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

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

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

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

Assembler source file `f1.s`:

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

Assembler source file `f2.s`:

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

Assembler source file `f3.s`:

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

Symbol control directives

Syntax	<pre>EXTERN <i>symbol</i> [, <i>symbol</i>] ... EXTWEAK <i>symbol</i> [, <i>symbol</i>] ... IMPORT <i>symbol</i> [, <i>symbol</i>] ... PUBLIC <i>symbol</i> [, <i>symbol</i>] ... PUBWEAK <i>symbol</i> [, <i>symbol</i>] ... REQUIRE <i>symbol</i></pre>
Parameters	<p><i>label</i> Label to be used as an alias for a C/C++ symbol.</p> <p><i>symbol</i> Symbol to be imported or exported.</p>
Description	<p>These directives control how symbols are shared between modules:</p>

Directive	Description
EXTERN, IMPORT	Imports an external symbol.

Table 15: Symbol control directives

Directive	Description
EXTWEAK	Imports an external symbol. The symbol can be undefined.
OVERLAY	Recognized but ignored.
PUBLIC	Exports symbols to other modules.
PUBWEAK	Exports symbols to other modules, multiple definitions allowed.
REQUIRE	Forces a symbol to be referenced.

Table 15: Symbol control directives (Continued)

Exporting symbols to other modules

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

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

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

Exporting symbols with multiple definitions to other modules

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

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

Importing symbols

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

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

Example

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

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

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

```

                                name    errorMessage
                                extern  print
                                public  err

                                section MYCODE:CODE(2)
                                arm

err                                adr     r0,msg
                                bl      print
                                bx      lr

msg                                data
                                dc8     "*** Error ***"
                                end

```

Mode control directives

Syntax

```

ARM
CODE
CODE16
CODE32
DATA
THUMB

```

Description

These directives provide control over the processor mode:

Directive	Description
ARM, CODE32	Subsequent instructions are assembled as 32-bit (Arm) instructions. Labels within a CODE32 area have bit 0 set to 0. Force 4-byte alignment.
CODE	Subsequent instructions are interpreted as Arm or Thumb instructions, depending on the setting of the assembler option <code>--arm</code> , <code>--cpu_mode</code> , or <code>--thumb</code> .

Table 16: Mode control directives

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

Table 16: Mode control directives

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.

Example

The following example shows how a Thumb entry to an Arm function can be implemented:

```

        name    modeChange
        section MYCODE:CODE(2)
        thumb

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

        nop                    ; For alignment only.
        arm

armEntry
    ; ...

    end

```

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

Syntax

```

ALIGN align [, value]
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.

<i>flag</i>	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.
<i>section</i>	The name of the section. The section name is a user-defined symbol that follows the rules described in <i>Symbols</i> , page 24.
<i>type</i>	The memory type, which can be either CODE, CONST, or DATA.
<i>value</i>	Byte value used for padding, default is zero.
<i>type-expr</i>	A constant expression identifying the ELF type of the section.
<i>flags-expr</i>	A constant expression identifying the ELF flags of the section.

Description

The section directives control how code and data are located. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 30.

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

Table 17: Section control directives

Directive	Description	Expression restrictions
SECTION	Begins an ELF section.	No external references Absolute
SECTION_TYPE	Sets ELF type and flags for a section.	
STACK	Begins a stack segment.	

Table 17: Section control directives (Continued)

Beginning a named absolute segment

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

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

Beginning a relocatable section

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

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

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

In the following example, the data following the first `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

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 value used for padding bytes 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.

This example starts a section, `MYDATA`, 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

```

Syntax          label = expr
                label ALIAS expr
                label ASSIGN expr

```

```

label DEFINE const_expr
label EQU expr
label SET expr
label SETA expr
label VAR expr

```

Parameters

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

Description

These directives are used for assigning values to symbols:

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

Table 18: Value assignment directives

Defining a temporary value

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

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

```

                name    table
cons           set     1

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

```

```

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

```

Defining a permanent local value

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

Use `EXTERN` to import symbols from other modules.

Defining a permanent global value

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

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

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

Conditional assembly directives

Syntax

```

ELSE
ELSEIF condition
ENDIF
IF condition

```

Parameters

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

Description

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.

Example

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

```
?add      macro    a,b,c
           if      _args == 2
           adds   a,a,#b
           elseif  _args == 3
           adds   a,b,#c
           endif
           endm

           name    addWithMacro
           section MYCODE:CODE(2)
           arm

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

           end
```

Macro processing directives**Syntax**

```
_args
ENDM
ENDR
EXITM
LOCAL symbol [,symbol] ...
name MACRO [argument] [,argument] ...
```



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

Parameters

actual Strings to be substituted.

argument Symbolic argument names.

expr An expression.

formal An argument into which each character of *actual* (REPTC) or each string of *actual* (REPTI) is substituted.

name The name of the macro.

symbol Symbols to be local to the macro.

Description

These directives allow user macros to be defined. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions*, page 30.

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

Table 19: Macro processing directives

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

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

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

The macro process consists of three distinct phases:

- 1 The assembler scans and saves macro definitions. The text between `MACRO` and `ENDM` is saved but not syntax checked. 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.

Defining a macro

You define a macro with the statement:

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

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

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

```

                                name    errMacro
                                extern  abort
errMac                          macro   text
                                bl      abort
                                data
                                dc8    text, 0
                                endm
```

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

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

The assembler expands this to:

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

end
```

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

The previous example could therefore be written as follows:

```
name     errMacro
extern  abort
errMac   macro  text
        bl      abort
        data
        dc8     \1,0
        endm
```

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

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

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

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

Note: It is illegal to redefine a macro.

Passing special characters

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

For example:

```

                                name    cmpMacro
cmp_reg                          macro  op
                                CMP    op
                                endm

```

The macro can be called using the macro quote characters:

```

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

```

You can redefine the macro quote characters with the `-M` command line option; see *-M*, page 54.

Predefined macro symbols

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

```

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

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

```

It generates this code:

```

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

```

Repeating statements

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

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

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

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

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

```

        name    reptc
        extern  plotc
        section MYCODE:CODE(2)

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

        end

```

This produces this code:

```

     9                                     name    reptc
    10                                     extern  plotc
    11                                     section MYCODE:CODE(2)
    12
    13             banner                  reptc  chr, "Welcome"
    14                                     movs   r0, #'chr'  ; Pass char as
parameter
    15                                     bl     plotc
    16                                     endr
    16.1 00000000 5700B0E3                movs   r0, #'W'   ; Pass char as
parameter
    16.2 00000004 .....                   bl     plotc
    16.3 00000008 6500B0E3                movs   r0, #'e'   ; Pass char
as
    16.4 0000000C .....                   bl     plotc
    16.5 00000010 6C00B0E3                movs   r0, #'l'   ; Pass char
as parameter.
    16.6 00000014 .....                   bl     plotc
    16.7 00000018 6300B0E3                movs   r0, #'c'   ; Pass char
as parameter.
    16.8 0000001C .....                   bl     plotc
    16.9 00000020 6F00B0E3                movs   r0, #'o'   ; Pass char
as parameter.
    16.10 00000024 .....                  bl     plotc
    16.11 00000028 6D00B0E3               movs   r0, #'m'   ; Pass char
as parameter.
    16.12 0000002C .....                  bl     plotc
    16.13 00000030 6500B0E3               movs   r0, #'e'   ; Pass char
as parameter.
    16.14 00000034 .....                  bl     plotc
    17
    18                                     end

```

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

```

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

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

        end

```

This produces this code:

```

     9                                name    repti
    10                                extern  a,b,c
    11                                section MYCODE:CODE(2)
    12
    13 00000000 0000B0E3 clearABC  movs   r0,#0
    14                                repti  location,a,b,c
    15                                ldr   r1,=location
    16                                str   r0,[r1]
    17                                endr
    17.1 00000004 10109FE5         ldr   r1,=a
    17.2 00000008 000081E5         str   r0,[r1]
    17.3 0000000C 0C109FE5         ldr   r1,=b
    17.4 00000010 000081E5         str   r0,[r1]
    17.5 00000014 08109FE5         ldr   r1,=c
    17.6 00000018 000081E5         str   r0,[r1]
    18
    19                                end

```

Coding inline for efficiency

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

This example outputs bytes from a buffer to a port:

```

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

ioPort  equ    0x0100

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

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

                                end

```


For efficiency we can recode this using a macro:

```

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

ioPort    equ    0x0100

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

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

           end

```

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

Listing control directives

Syntax	<code>COL</code> <i>columns</i>
	<code>LSTCND</code> {+ -}
	<code>LSTCOD</code> {+ -}
	<code>LSTEXP</code> {+ -}
	<code>LSTMAC</code> {+ -}
	<code>LSTOUT</code> {+ -}
	<code>LSTPAG</code> {+ -}

```
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

Description

These directives provide control over the assembler list file:

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

Table 20: Listing control directives

Turning the listing on or off

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

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

To disable the listing of a debugged section of program:

```
lstout-
; This section has already been debugged.
lstout+
; This section is currently being debugged.
end
```

Listing conditional code and strings

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

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

Use `LSTCOD-` to 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.

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

```

                                name    lstcndTest
                                extern  print
                                section FLASH:CODE(2)

debug    set    0
         if    debug
         bl    print
         endif

         lstcnd+
begin2   if    debug
         bl    print
         endif

         end

```

This generates the following listing:

```

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

```

Controlling the listing of macros

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

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

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

```

        name    lstmacTest
        extern  memLoc
        section FLASH:CODE(2)

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

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

begin   dec2    memLoc
        lstexp-
        inc2    memLoc
        bx     lr

; Restore default values for
; listing control directives.

        lstmac-
        lstexp+

        end

```

This produces the following output:

```

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

```

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.

C-style preprocessor directives

Syntax

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

Parameters

<i>condition</i>	An absolute assembler expression, see <i>Expressions, operands, and operators</i> , page 22. The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true. The C preprocessor operator <code>defined</code> can be used.
<i>filename</i>	Name of file to be included or referred.
<i>line-no</i>	Source line number.
<i>message</i>	Text to be displayed.
<i>symbol</i>	Preprocessor symbol to be defined, undefined, or tested.
<i>text</i>	Value to be assigned.

Description

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

These C-language preprocessor directives are available:

Directive	Description
<code>#define</code>	Assigns a value to a preprocessor symbol.
<code>#elif</code>	Introduces a new condition in an <code>#if...#endif</code> block.
<code>#else</code>	Assembles instructions if a condition is false.
<code>#endif</code>	Ends an <code>#if</code> , <code>#ifdef</code> , or <code>#ifndef</code> block.
<code>#error</code>	Generates an error.
<code>#if</code>	Assembles instructions if a condition is true.
<code>#ifdef</code>	Assembles instructions if a preprocessor symbol is defined.
<code>#ifndef</code>	Assembles instructions if a preprocessor symbol is undefined.
<code>#include</code>	Includes a file.
<code>#message</code>	Generates a message on standard output.
<code>#pragma</code>	This directive is recognized but ignored.
<code>#undef</code>	Undefines a preprocessor symbol.

Table 21: C-style preprocessor directives

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.

This example defines the labels `tweak` and `adjust`. If `adjust` is defined, then register 16 is decremented by an amount that depends on `adjust`, for example 30 when `adjust` is 3.

```

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

#define         tweak  1
#define         adjust 3

calibrate      ldr     r0,calibrationConstant
#ifdef         tweak
#if            adjust==1
                subs   r0,r0,#4
#elif         adjust==2
                subs   r0,r0,#20
#elif         adjust==3
                subs   r0,r0,#30
#endif
#endif
                /* ifdef tweak */
                str     r0,calibrationConstant
                bx      lr

                end

```


Including 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 51.

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

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

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

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

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

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

xchRegs  xch     r0,r1,r2
          bx     lr

          end
```

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.

Changing the source line numbers

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

Comments in C-style preprocessor directives

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

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

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

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

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

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

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

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

                module misplacedComment2
                section MYCONST:CONST(2)

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

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

                end
```

Data definition or allocation directives

Syntax	<pre> DC8 <i>expr</i> [, <i>expr</i>] ... DC16 <i>expr</i> [, <i>expr</i>] ... DC24 <i>expr</i> [, <i>expr</i>] ... DC32 <i>expr</i> [, <i>expr</i>] ... DCB <i>expr</i> [, <i>expr</i>] ... DCD <i>expr</i> [, <i>expr</i>] ... DCW <i>expr</i> [, <i>expr</i>] ... DF32 <i>value</i> [, <i>value</i>] ... DF64 <i>value</i> [, <i>value</i>] ... DS <i>count</i> DS8 <i>count</i> DS16 <i>count</i> DS24 <i>count</i> DS32 <i>count</i> </pre>
Parameters	<p><i>count</i> A valid absolute expression specifying the number of elements to be reserved.</p> <p><i>expr</i> 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.</p> <p><i>value</i> A valid absolute expression or floating-point constant.</p>
Description	<p>These directives define values or reserve memory.</p> <p>Use DC8, DC16, DC24, DC32, DCB, DCD, DCW, DF32, or DF64 to create a constant, which means an area of bytes is reserved big enough for the constant.</p> <p>Use DS8, DS16, DS24, or DS32 to reserve a number of uninitialized bytes.</p> <p>For information about the restrictions that apply when using a directive in an expression, see <i>Expression restrictions</i>, page 30.</p> <p>The column <i>Alias</i> in the following table shows the Arm Limited directive that corresponds to the IAR Systems directive.</p>

Directive	Alias	Description
DC8	DCB	Generates 8-bit constants, including strings.
DC16	DCW	Generates 16-bit constants.
DC24		Generates 24-bit constants.
DC32	DCD	Generates 32-bit constants.

Table 22: Data definition or allocation directives

Directive	Alias	Description
DF32		Generates 32-bit floating-point constants.
DF64		Generates 64-bit floating-point constants.
DS8	DS	Allocates space for 8-bit integers.
DS16		Allocates space for 16-bit integers.
DS24		Allocates space for 24-bit integers.
DS32		Allocates space for 32-bit integers.

Table 22: Data definition or allocation directives (Continued)

Generating a lookup table

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

```

                                module sumTableAndIndex
                                section MYDATA:CONST
                                data

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

                                section MYCODE:CODE(2)
                                arm
count      set      0

addTable   movs     r0,#0
           ldr      r1,=table

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

count      set      count + 1
           endr

           bx      lr

           end

```

Defining strings

To define a string:

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

To define a string which includes a trailing zero:

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

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

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

Reserving space

To reserve space for 10 bytes:

```
table DS8 10
```

Assembler control directives

Syntax	<i>\$filename</i> <i>/*comment*/</i> <i>//comment</i> CASEOFF CASEON INCLUDE <i>filename</i> LTOrg RADIX <i>expr</i>	
Parameters	<i>comment</i> <i>expr</i> <i>filename</i>	Comment ignored by the assembler. Default base; default 10 (decimal). Name of file to be included. The \$ character must be the first character on the line.
Description	These directives provide control over the operation of the assembler. For information about the restrictions that apply when using a directive in an expression, see <i>Expression restrictions</i> , page 30.	

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

Table 23: Assembler control directives

Use `$` to insert the contents of a file into the source file at a specified point. This is an alias for `#include`, see the section *Including source files* under *Assembler control directives*, page 117.

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 `LORG` 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 133.

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.

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

```

module caseSensitivity1
section MYCODE:CODE(2)

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

```

The following will generate a duplicate label error:

```

module caseSensitivity2

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

```

Including a source file

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

```

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

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

```

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

```

name includeFile
section MYCODE:CODE(2)

; Standard macro definitions.
$Macros.inc

xchRegs xch r0,r1,r2
bx lr

end

```

Defining comments

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

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

See also *C-style preprocessor directives*, page 110, specifically *Comments in C-style preprocessor directives*.

Changing the base

To set the default base to 16:

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

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

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

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

end
```

Function directives

Syntax	<code>CALL_GRAPH_ROOT <i>function</i> [,<i>category</i>]</code>
Parameters	<p><i>function</i> The function, a symbol.</p> <p><i>category</i> An optional call graph root category, a string.</p>
Description	<p>Use this directive to specify that, for stack usage analysis purposes, the function <i>function</i> is a call graph root. You can also specify an optional category, a quoted string.</p> <p>The compiler will generate this directive in assembler list files, when needed.</p>

Example `CALL_GRAPH_ROOT my_interrupt, "interrupt"`

See also *Call frame information directives for stack usage analysis*, page 127, for information about CFI directives required for stack usage analysis.

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

Call frame information directives for names blocks

Syntax

Names block directives:

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

Parameters

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

Description

Use these directives to define a names block:

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

Table 24: Call frame information directives names block

Example

Examples of using CFI directives, page 40

See also

Tracking call frame usage, page 32

Call frame information directives for common blocks

Syntax

Common block directives:

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

Parameters

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

commonblock The name of a previously defined common block.

<i>dataalignfactor</i>	The smallest common factor of all frame sizes. If the stack grows toward higher addresses, the factor is negative; if it grows toward lower addresses, the factor is positive. 1 is the default, but a larger value reduces the produced call frame information in size. The possible ranges are -256 to -1 and 1 to 256 .
<i>name</i>	The name of the block.
<i>namesblock</i>	The name of a previously defined names block.
<i>resource</i>	The name of a resource.
<i>type</i>	The memory type, such as CODE, CONST or DATA. In addition, any of the segment memory types supported by the IAR ILINK Linker. It is only used for denoting an address space.

Description

Use these directives to define a common block:

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

Table 25: Call frame information directives common block

In addition to these directives you might also need the call frame information directives for specifying rules or CFI expressions for resources and CFAs, see *Call frame information directives for tracking resources and CFAs*, page 125.

Example

Examples of using CFI directives, page 40

See also

Tracking call frame usage, page 32

Call frame information directives for data blocks

Syntax

```
CFI BLOCK name USING commonblock
CFI ENDBLOCK name
CFI { NOFUNCTION | FUNCTION label }
CFI { INVALID | VALID }
```

```
CFI { REMEMBERSTATE | RESTORESTATE }
CFI PICKER
CFI CONDITIONAL label [, label] ...
```

Parameters

commonblock The name of a previously defined common block.

label A function label.

name The name of the block.

Description

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

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

Table 26: Call frame information directives for data blocks

In addition to these directives you might also need the call frame information directives for specifying rules or CFI expressions for resources and CFAs, see *Call frame information directives for tracking resources and CFAs*, page 125.

Example

Examples of using CFI directives, page 40

See also

Tracking call frame usage, page 32

Call frame information directives for tracking resources and CFAs

Syntax

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

Parameters

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

cfiexpr A CFI expression, which can be one of these:

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

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

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

resource The name of a resource.

Unary operators

Overall syntax: *OPERATOR*(*operand*)

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

Table 27: Unary operators in CFI expressions

Binary operators

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

CFI operator	Operands	Description
ADD	<i>cfiexpr</i> , <i>cfiexpr</i>	Addition

Table 28: Binary operators in CFI expressions

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

Table 28: Binary operators in CFI expressions (Continued)

Ternary operators

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

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

Table 29: Ternary operators in CFI expressions

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

Table 29: Ternary operators in CFI expressions (Continued)

Description

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

Directive	Description
CFI <i>cfa</i>	Declares the value of a CFA.
CFI <i>resource</i>	Declares the value of a resource.

Table 30: Call frame information directives for tracking resources and CFAs

Example

Examples of using CFI directives, page 40

See also

Tracking call frame usage, page 32

Call frame information directives for stack usage analysis

Syntax

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

Parameters

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

Description

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

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

Table 31: Call frame information directives for stack usage analysis

See also

Tracking call frame usage, page 32

The *IAR C/C++ Development Guide for Arm* for information about stack usage analysis.

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.

Note: In Thumb mode (and CODE16), the syntax `LDR register, =expression` can, for values from 0 to 255, be translated into a MOV instruction. This instruction modifies the program status register.

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

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

Table 32: Pseudo-instructions

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

Table 32: Pseudo-instructions (Continued)

* Deprecated. Use THUMB instead.

Descriptions of pseudo-instructions

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

ADR (ARM)

Syntax

`ADR{condition} register,expression`

Parameters

<code>{condition}</code>	Can be one of the following: EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE, and AL.
<code>register</code>	The register to load.
<code>expression</code>	A program location counter-relative expression that evaluates to an address that is not word-aligned within the range -247 to +263 bytes, or a word-aligned address within the range -1012 to +1028 bytes. Unresolved expressions (for example expressions that contain external labels, or labels in other sections) must be within the range -247 to +263 bytes.

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

```

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

thumb
thumbLabel ; ...

end

```

ADR (CODE16)

Syntax 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    thumbAdr
section MYCODE:CODE(2)
thumb
adr     r0,dataLabel    ; Becomes "add r0,pc,#4".
add     r0,r0,r1
bx      lr

data
alignr 2
dataLabel dc32 0xABCD19

end
    
```

See also

ADR (CODE16), page 131 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	<pre> name armAdrL section MYCODE:CODE(2) arm adr1 r1,label+0x2345 ; Becomes "add r1,pc,#0x45" ; and "add r1,r1,#0x2300" data label dc32 0 end </pre>
---------	--

ADRL (THUMB)

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

LDR (ARM)

Syntax	<pre> LDR{condition} register,=expression1 or LDR{condition} register,expression2 </pre>
Parameters	<p><i>condition</i> An optional condition code.</p> <p><i>register</i> The register to load.</p> <p><i>expression1</i> Any 32-bit expression.</p> <p><i>expression2</i> A program location counter-relative expression in the range -4087 to +4103 from the program location counter.</p>

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.

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 0xFFEEDDCC into r2.

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

See also The `LTORG` directive in the section *Assembler control directives*, page 117.

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. Note that the first form can be translated into a `MOVSV` instruction, which modifies the program status register.

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	<code>LDR{condition} register,=expression</code>
Parameters	<p><i>condition</i> An optional condition code if the instruction is placed after an IT instruction.</p> <p><i>register</i> The register to load.</p> <p><i>expression</i> Any 32-bit expression.</p>
Description	<p>Similar to the 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.</p> <p>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.</p> <p>By specifying a 32-bit version explicitly with the LDR.W instruction, a 32-bit instruction is always generated.</p> <p>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.</p> <p>As for LDR (CODE16), the 16-bit variant can be translated into a MOVS instruction, which modifies the program status register.</p> <p>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.</p>
Example	<pre>name thumbLdr extern extLabel</pre>

```

                                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 *ADR (CODE16)*, page 131 if only 16-bit Thumb instructions are available.

MOV (CODE16)

Syntax

MOV Rd, Rs

Parameters

Rd The destination register.
 Rs The source register.

Description

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

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

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

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

Example

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

MOV32 (THUMB)

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

NOP (ARM)

Syntax	<code>NOP</code>
Description	<p><code>NOP</code> generates the preferred Arm no-operation code:</p> <pre>MOV r0, r0</pre> <p>Note: <code>NOP</code> is not a pseudo-instruction in architecture versions that include a <code>NOP</code> instruction (Armv6K, Armv6T2, Armv7, Armv8-M).</p>

NOP (CODE16)

Syntax	<code>NOP</code>
Description	<p><code>NOP</code> generates the preferred Thumb no-operation code:</p> <pre>MOV r8, r8</pre> <p>Note: <code>NOP</code> is not a pseudo-instruction in architecture versions that include a <code>NOP</code> instruction (Armv6T2, Armv7, Armv8-M).</p>

Assembler diagnostics

The following pages describe the format of the diagnostic messages and explains how diagnostic messages are divided into different levels of severity.

Message format

All diagnostic messages are displayed on the screen, and printed in the optional list file.

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

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

Severity levels

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

OPTIONS FOR DIAGNOSTICS

There are two assembler options for diagnostics. You can:

- Disable or enable all warnings, ranges of warnings, or individual warnings, see *-w*, page 61
- Set the number of maximum errors before the assembly stops, see *-E*, page 49.

ASSEMBLER WARNING MESSAGES

Assembler 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.

ASSEMBLER ERROR MESSAGES

Assembler error messages are produced when the assembler finds a construct which violates the language rules.

ASSEMBLER FATAL ERROR MESSAGES

Assembler 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

Assembler 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 Arm Limited.

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 can cause problems without generating errors or warnings. Be extra careful when you use such labels in situations other than jumps.

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

THUMB CODE LABELS

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

Example

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

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

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

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

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

thumb
T      nop
end
```

Alternative register names

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

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

Table 33: Alternative register names

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

Table 33: Alternative register names (Continued)

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

Alternative mnemonics

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

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

Table 34: Alternative mnemonics

Alternative mnemonic	Assembler mnemonic
BNVS	BVC
CMN{cond}S	CMN{cond}
CMP{cond}S	CMP{cond}
EORS	EOR
LSL	LSL
LSRS	LSR
MOVS	MOV
MULS	MUL
MVNS	MVN
NEGS	NEG
ORRS	ORR
RORS	ROR
SBCS	SBC
SUBS	SUB
TEQ{cond}S	TEQ{cond}
TST{cond}S	TST{cond}

Table 34: Alternative mnemonics (Continued)

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

Operator synonyms

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

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

Table 35: Operator synonyms

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

Table 35: Operator synonyms (Continued)

Note: In some cases, assembler operators and operator synonyms have different precedence levels. For further descriptions of the operators, see the chapter *Assembler operators*, page 63.

Warning messages

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

THE FIRST REGISTER OPERAND OMITTED

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

THE FIRST REGISTER OPERAND DUPLICATED

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

Example of incorrect code:

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

Example of correct code:

```
MUL R0, R1
```

IMMEDIATE #0 OMITTED IN LOAD/STORE

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

Example of incorrect code:

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

Example of correct code:

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


A

- absolute expressions 29
- ADD (CFI operator) 125
- addition (assembler operator) 67
- address field, in assembler list file 31
- addresses, loading into a register 130–133
- ADR (ARM) (pseudo-instruction) 130
- ADR (CODE16) (pseudo-instruction). 131
- ADR (THUMB) (pseudo-instruction). 131
- ADRL (ARM) (pseudo-instruction) 132
- ADRL (THUMB) (pseudo-instruction). 133
- ALIAS (assembler directive) 94
- ALIGN (assembler directive) 91
- alignment, of sections 93
- ALIGNRAM (assembler directive). 91
- ALIGNROM (assembler directive). 91
- :AND: (assembler operator) 70
- AND (CFI operator) 126
- _args (assembler directive). 97
- _args (predefined macro symbol) 100
- arm (assembler option). 46
- ARMASM assembler 141
- __ARMVFP__ (predefined symbol). 27
- __ARMVFP_D16__ (predefined symbol) 27
- __ARMVFP_FP16__ (predefined symbol). 27
- __ARMVFP_SP__ (predefined symbol). 27
- __ARM_ADVANCED_SIMD__ (predefined symbol) 25
- __ARM_ARCH (predefined symbol) 25
- __ARM_ARCH_ISA_ARM (predefined symbol) 25
- __ARM_ARCH_ISA_THUMB (predefined symbol). 26
- __ARM_ARCH_PROFILE (predefined symbol) 26
- __ARM_BIG_ENDIAN (predefined symbol) 26
- __ARM_FEATURE_CMSE (predefined symbol) 26
- __ARM_FEATURE_CRC32 (predefined symbol). 26
- __ARM_FEATURE_CRYPT0 (predefined symbol). 26
- __ARM_FEATURE_DIRECTED_ROUNDING (predefined symbol). 26
- __ARM_FEATURE_DSP (predefined symbol) 26
- __ARM_FEATURE_FMA (predefined symbol) 26
- __ARM_FEATURE_IDIV (predefined symbol) 26
- __ARM_FEATURE_NUMERIC_MAXMIN (predefined symbol). 26
- __ARM_FP (predefined symbol) 27
- __ARM_MEDIA__ (predefined symbol) 27
- __ARM_MPCORE__ (predefined symbol) 27
- __ARM_NEON (predefined symbol) 27
- __ARM_NEON_FP (predefined symbol). 27
- __ARM_PROFILE_M__ (predefined symbol). 27
- ASCII character constants. 23
- assembler control directives 117
- assembler diagnostics 139
- assembler directives
 - assembler control 117
 - CFI directives for common blocks 122
 - CFI directives for data blocks 123
 - CFI directives for names blocks. 121
 - CFI directives for tracking resources and CFAs. 125
 - CFI for stack usage analysis) 127
 - conditional assembly 95
 - See also* C-style preprocessor directives
 - C-style preprocessor 110
 - data definition or allocation 115
 - function 120
 - list file control 105
 - macro processing 96
 - module control 83
 - segment control 90
 - summary 79
 - symbol control 86
 - value assignment 93
- assembler environment variables 20
- assembler error messages 140
- assembler expressions. 22
- assembler instructions. 21
- BX 89
- assembler invocation syntax 19
- assembler labels 24
- format of 21

in Thumb code	141
assembler list files	
address field	31
comments	118
conditional code and strings	107
cross-references	
generating (LSTXRF)	109
generating (-x)	62
data field	31
enabling and disabling (LSTOUT)	106
filename, specifying (-I)	53
generated lines, controlling (LSTREP)	109
generating (-L)	53
header section, omitting (-N)	55
#include files, specifying (-i)	52
lines per page, specifying (-p)	57
macro execution information, including (-B)	46
macro-generated lines, controlling	108
symbol and cross-reference table	31
tab spacing, specifying	60
using directives to format	109
assembler macros	
arguments, passing to	100
defining	98
generated lines, controlling in list file	108
inline routines	103
predefined symbol	100
quote characters, specifying	54
special characters, using	99
assembler object file, specifying filename	56
assembler operators	63
in expressions	22
precedence	63
assembler options	
passing to assembler	20
command line, setting	43
extended command file, setting	44
specifying parameters	43
summary	44

assembler output, including debug information	58
assembler pseudo-instructions	129
assembler source files, including	113, 119
assembler source format	21
assembler symbols	24
exporting	87
importing	87
in relocatable expressions	29
predefined	25
undefining	61
assembler warning messages	139
assembling, invocation syntax	19
assembly messages format	139
assembly warning messages	
disabling	61
ASSIGN (assembler directive)	94
assumptions (programming experience)	13

B

-B (assembler option)	46
bitwise AND (assembler operator)	70
bitwise exclusive OR (assembler operator)	71
bitwise NOT (assembler operator)	71
bitwise OR (assembler operator)	71
bold style, in this guide	15
__BUILD_NUMBER__ (predefined symbol)	28
BX (assembler instruction)	89
byte order	28
BYTE1 (assembler operator)	73
BYTE2 (assembler operator)	73
BYTE3 (assembler operator)	73
BYTE4 (assembler operator)	74

C

-c (assembler option)	47
call frame information directives	121–123, 125, 127
CALL_GRAPH_ROOT (assembler directive)	120

- case sensitive user symbols 58
- case sensitivity, controlling 118
- CASEOFF (assembler directive) 118
- CASEON (assembler directive) 118
- CFA, CFI directives for tracking 125
- CFI BASEADDRESS (assembler directive) 122
- CFI BLOCK (assembler directive) 124
- CFI cfa (assembler directive) 127
- CFI CODEALIGN (assembler directive) 123
- CFI COMMON (assembler directive) 123
- CFI CONDITIONAL (assembler directive) 124
- CFI DATAALIGN (assembler directive) 123
- CFI DEFAULT (assembler directive) 123
- CFI directives for common blocks 122
- CFI directives for data blocks 123
- CFI directives for names blocks 121
- CFI directives for stack usage analysis 127
- CFI directives for tracking resources and CFAs 125
- CFI ENDBLOCK (assembler directive) 124
- CFI ENDCOMMON (assembler directive) 123
- CFI ENDNAMES (assembler directive) 122
- CFI expressions 39
- CFI FRAMECELL (assembler directive) 122
- CFI FUNCALL (assembler directive) 128
- CFI FUNCTION (assembler directive) 124
- CFI INDIRECTCALL (assembler directive) 128
- CFI INVALID (assembler directive) 124
- CFI NAMES (assembler directive) 122
- CFI NOCALLS (assembler directive) 128
- CFI NOFUNCTION (assembler directive) 124
- CFI PICKER (assembler directive) 124
- CFI REMEMBERSTATE (assembler directive) 124
- CFI RESOURCE (assembler directive) 122
- CFI resource (assembler directive) 127
- CFI RESOURCEPARTS (assembler directive) 122
- CFI RESTORESTATE (assembler directive) 124
- CFI RETURNADDRESS (assembler directive) 123
- CFI STACKFRAME (assembler directive) 122
- CFI TAILCALL (assembler directive) 128
- CFI VALID (assembler directive) 124
- CFI VIRTUALRESOURCE (assembler directive) 122
- character constants, ASCII 23
- cmse (assembler option) 47
- CODE (assembler directive) 88
- CODE16 (assembler directive) 89
- CODE32 (assembler directive) 88
- COL (assembler directive) 106
- command line error messages, assembler 139
- command line options 43
 - part of invocation syntax 19
 - passing 20
 - typographic convention 15
- command line, extending 50
- command prompt icon, in this guide 15
- comments
 - in assembler list file 118
 - in assembler source code 21
 - in C-style preprocessor directives 114
 - multi-line, using with assembler directives 120
- common block (call frame information) 33
- common blocks, CFI directives for 122
- common block, defining 35
- COMPLEMENT (CFI operator) 125
- computer style, typographic convention 14
- conditional assembly directives 95
 - See also* C-style preprocessor directives
- conditional code and strings, listing 107
- constants
 - default base of 118
 - integer 22
- conventions, used in this guide 14
- copyright notice 2
- __CORE__ (predefined symbol) 28
- cpu (assembler option) 47
- cpu_mode (assembler option) 48
- CPU, defining in assembler. *See* processor configuration
- CRC, in assembler list file 31

cross-references, in assembler list file	
generating (LSTXRF)	109
generating (-x)	62
current time/date (assembler operator)	74
C-style preprocessor directives	110
C++ terminology	14

D

-D (assembler option)	48
data allocation directives	115
data block (call frame information)	34
data blocks, CFI directives for	123
data definition directives	115
data field, in assembler list file	31
DATA (assembler directive)	89
data, defining in Thumb code section	89
__DATE__ (predefined symbol)	28
DATE (assembler operator)	74
DCB (assembler directive)	115
DCD (assembler directive)	115
DCW (assembler directive)	115
DC8 (assembler directive)	115
DC16 (assembler directive)	115
DC24 (assembler directive)	115
DC32 (assembler directive)	115
debug information, including in assembler output	58
default base, for constants	118
#define (assembler directive)	111
DEFINE (assembler directive)	94
defining a common block	35
defining a names block	34
DF32 (assembler directive)	116
DF64 (assembler directive)	116
diagnostic messages	
options for	139
diagnostics	139
directives. <i>See</i> assembler directives	
disclaimer	2

DIV (CFI operator)	126
division (assembler operator)	68
DLIB	
naming convention	16
document conventions	14
DS (assembler directive)	116
DS8 (assembler directive)	116
DS16 (assembler directive)	116
DS24 (assembler directive)	116
DS32 (assembler directive)	116

E

-E (assembler option)	49
-e (assembler option)	49
edition, of this guide	2
efficient coding techniques	31
#elif (assembler directive)	111
#else (assembler directive)	111
END (assembler directive)	84
--endian (assembler option)	49
#endif (assembler directive)	111
ENDM (assembler directive)	97
ENDR (assembler directive)	97
environment variables	
assembler	20
:EOR: (assembler operator)	71
EQ (CFI operator)	126
EQU (assembler directive)	94
equal (assembler operator)	69
#error (assembler directive)	111
error messages	
format	139
maximum number, specifying	49
#error, using to display	113
EVEN (assembler directive)	91
EXITM (assembler directive)	97
experience, programming	13
expressions	22

extended command line file (extend.xcl)	44, 50
EXTERN (assembler directive)	86
EXTWEAK (assembler directive)	87

F

-f (assembler option)	44, 50
false value, in assembler expressions	23
fatal errors	140
__FILE__ (predefined symbol)	28
file extensions. <i>See</i> filename extensions	
file types	
extended command line	44, 50
#include, specifying path	51
filename extensions	
xcl	44, 50
filenames, specifying for assembler object file	56–57
first byte (assembler operator)	73
floating-point constants	23
floating-point coprocessor, defining in assembler	50
formats	
assembler source code	21
diagnostic messages	139
in list files	30
fourth byte (assembler operator)	74
--fpu (assembler option)	50
FRAME (CFI operator)	126
function directives	120

G

-G (assembler option)	51
-g (assembler option)	51
GE (CFI operator)	126
global value, defining	95
greater than or equal (assembler operator)	70
greater than (assembler operator)	69
GT (CFI operator)	126

H

header files, SFR	31
header section, omitting from assembler list file	55
high byte (assembler operator)	74
high word (assembler operator)	75
HIGH (assembler operator)	74
HWRD (assembler operator)	75

I

-I (assembler option)	51
__IAR_SYSTEMS_ASM__ (predefined symbol)	28
__IASMARM__ (predefined symbol)	28
icons, in this guide	15
#if (assembler directive)	111
IF (CFI operator)	127
#ifdef (assembler directive)	111
#ifndef (assembler directive)	111
IMPORT (assembler directive)	86
#include files	52
#include files, specifying	51
#include (assembler directive)	111
include files, disabling search for	51
include paths, specifying	51
INCLUDE (assembler directive)	118
inline coding, using macros	103
installation directory	14
integer constants	22
internal errors, assembler	140
invocation syntax	19
italic style, in this guide	15

J

-j (assembler option)	52
---------------------------------	----

L

-L (assembler option)	53
-l (assembler option)	53
labels. <i>See</i> assembler labels	
:LAND: (assembler operator)	70
LDR (ARM) (pseudo-instruction).	133
LDR (CODE16) (pseudo-instruction).	134
LDR (THUMB) (pseudo-instruction)	135
LE (CFI operator)	126
--legacy (assembler option).	54
:LEOR: (assembler operator)	78
less than or equal (assembler operator).	68
less than (assembler operator).	68
LIBRARY (assembler directive).	82
lightbulb icon, in this guide.	15
__LINE__ (predefined symbol)	28
lines per page, in assembler list file	57
linker options	
typographic convention	15
list file format	30
body	31
CRC	31
header	30
symbol and cross reference	
list files	
control directives for	105
controlling contents of (-c).	47
cross-references, generating (-x)	62
filename, specifying (-l).	53
generating (-L).	53
header section, omitting (-N).	55
#include files, specifying (-i)	52
literal pool.	56, 134
LITERAL (CFI operator)	125
__LITTLE_ENDIAN__ (predefined symbol).	28
:LNOT: (assembler operator)	72
LOAD (CFI operator).	127
local value, defining	95

LOCAL (assembler directive).	97
logical AND (assembler operator)	70
logical exclusive OR (assembler operator)	78
logical NOT (assembler operator).	72
logical OR (assembler operator)	72
logical shift left (assembler operator)	72
logical shift right (assembler operator)	73
:LOR: (assembler operator)	72
low byte (assembler operator).	75
low register values, moving	136–137
low word (assembler operator)	75
LOW (assembler operator)	75
LSHIFT (CFI operator).	126
LSTCND (assembler directive).	106
LSTCOD (assembler directive).	106
LSTEXP (assembler directives)	106
LSTMAC (assembler directive)	106
LSTOUT (assembler directive).	106
LSTPAG (assembler directive).	106
LSTREP (assembler directive)	106
LSTXRF (assembler directive).	106
LT (CFI operator).	126
LTORG (assembler directive).	118
LWRD (assembler operator).	75

M

-M (assembler option).	54
macro execution information, including in list file	46
macro processing directives	96
macro quote characters	99
specifying	54
MACRO (assembler directive)	97
macros. <i>See</i> assembler macros	
memory, reserving space in	115
#message (assembler directive).	111
messages, excluding from standard output stream	58
migration to the IAR Assembler for Arm	141
alternative mnemonics	143

alternative register names	142
operator synonyms	144
warning messages	145
:MOD: (assembler operator)	71
MOD (CFI operator)	126
module consistency	85
module control directives	83
modules, beginning	84
MOV (CODE16) (pseudo-instruction)	136
MOV (THUMB) (pseudo-instruction)	137
MUL (CFI operator)	126
multiplication (assembler operator)	66

N

-N (assembler option)	55
NAME (assembler directive)	84
names block (call frame information)	33
names blocks, CFI directives for	121
names block, defining	34
naming conventions	15
NE (CFI operator)	126
NOP (ARM) (pseudo-instruction)	137
NOP (CODE16) (pseudo-instruction)	137
:NOT: (assembler operator)	71
not equal (assembler operator)	69
NOT (CFI operator)	125
--no_dwarf3_cfi (assembler option)	55
--no_it_verification (assembler option)	55
--no_literal_pool (assembler option)	55
--no_path_in_file_macros (assembler option)	56
no-operation code, generating	137

O

-O (assembler option)	56
-o (assembler option)	57
ODD (assembler directive)	91

operands	
format of	21
in assembler expressions	22
operations, format of	21
operation, silent	58
operators. <i>See</i> assembler operators	
option summary	44
:OR: (assembler operator)	71
OR (CFI operator)	126
OVERLAY (assembler directive)	87

P

-p (assembler option)	57
PAGE (assembler directive)	106
PAGSIZ (assembler directive)	106
pair, of registers	25
parameters	
specifying	43
typographic convention	15
part number, of this guide	2
#pragma (assembler directive)	111
precedence, of assembler operators	63
predefined register symbols	25
predefined symbols	25
in assembler macros	100
undefining	61
preprocessor symbols	
defining and undefining	111
defining on command line	48
prerequisites (programming experience)	13
program location counter (PLC)	24
program modules, beginning	84
PROGRAM (assembler directive)	84
programming experience, required	13
programming hints	31
pseudo-instructions	129
PUBLIC (assembler directive)	87
publication date, of this guide	2

PUBWEAK (assembler directive) 87

R

-r (assembler option) 58
RADIX (assembler directive) 118
reference information, typographic convention. 15
registered trademarks 2
registers 25
 alternative names of 142
relocatable expressions 29
repeating statements 101
REPT (assembler directive) 97
REPTC (assembler directive) 97
REPTI (assembler directive) 97
REQUIRE (assembler directive) 87
resources, CFI directives for tracking 125
RSEG (assembler directive) 91
RSHIFTA (CFI operator) 126
RSHIFTL (CFI operator) 126
RTMODEL (assembler directive) 84
rules, in CFI directives 37
runtime model attributes, declaring 85

S

-S (assembler option) 58
-s (assembler option) 58
second byte (assembler operator) 73
SECTION (assembler directive) 92
sections
 aligning 93
 beginning 92
SECTION_TYPE (assembler directive) 92
segment begin (assembler operator) 75
segment control directives 90
segment end (assembler operator) 76
segment size (assembler operator) 76
SET (assembler directive) 94

SETA (assembler directive) 94
SFB (assembler operator) 75
SFE (assembler operator) 76
SFR. *See* special function registers
:SHL: (assembler operator) 72
:SHR: (assembler operator) 73
silent operation, specifying in assembler. 58
simple rules, in CFI directives 37
SIZEOF (assembler operator) 76
source files
 example of including 119
 including 113
source format, assembler 21
source line numbers, changing 114
--source_encoding (assembler option) 59
special function registers. 31
stack usage analysis, CFI directives for 127
STACK (assembler directive) 92
standard input stream (stdin), reading from. 51
standard output stream, disabling messages to 58
statements, repeating. 101
SUB (CFI operator) 126
subtraction (assembler operator) 68
--suppress_vfe_header (assembler option) 59
symbol and cross-reference table, in assembler list file . . . 31
 See also Include cross-reference
symbol control directives 86
symbols
 See also assembler symbols
 exporting to other modules 87
 predefined, in assembler 25
 predefined, in assembler macro 100
 user-defined, case sensitive 58
system include files, disabling search for 51
--system_include_dir (assembler option) 60

T

-t (assembler option) 60

tab spacing, specifying in assembler list file	60
target core, specifying. <i>See</i> processor configuration	
temporary values, defining	94
terminology.	14
third byte (assembler operator)	73
--thumb (assembler option)	60
THUMB (assembler directive)	89
__TID__ (predefined symbol)	28
__TIME__ (predefined symbol)	28
time-critical code	103
tools icon, in this guide	15
trademarks	2
true value, in assembler expressions	23
typographic conventions	14

U

-U (assembler option)	61
UGT (assembler operator)	77
ULT (assembler operator)	77
UMINUS (CFI operator)	125
unary minus (assembler operator)	67
unary plus (assembler operator)	67
#undef (assembler directive)	111
unsigned greater than (assembler operator)	77
unsigned less than (assembler operator)	77
user symbols, case sensitive	58

V

value assignment directives	93
values, defining.	115
VAR (assembler directive)	94
__VER__ (predefined symbol)	28
version	
of this guide	2

W

-w (assembler option)	61
warnings	139
disabling	61
warnings icon, in this guide	15

X

-x (assembler option)	62
xcl (filename extension)	44, 50
XOR (assembler operator)	78
XOR (CFI operator)	126

Symbols

_args (assembler directive)	97
_args (predefined macro symbol)	100
__ARMVFP_D16__ (predefined symbol)	27
__ARMVFP_FP16__ (predefined symbol)	27
__ARMVFP_SP__ (predefined symbol)	27
__ARMVFP__ (predefined symbol)	27
__ARM_ADVANCED_SIMD__ (predefined symbol)	25
__ARM_ARCH (predefined symbol)	25
__ARM_ARCH_ISA_ARM (predefined symbol)	25
__ARM_ARCH_ISA_THUMB (predefined symbol)	26
__ARM_ARCH_PROFILE (predefined symbol)	26
__ARM_BIG_ENDIAN (predefined symbol)	26
__ARM_FEATURE_CMSE (predefined symbol)	26
__ARM_FEATURE_CRC32 (predefined symbol)	26
__ARM_FEATURE_CRYPT (predefined symbol)	26
__ARM_FEATURE_DIRECTED_ROUNDING (predefined symbol)	26
__ARM_FEATURE_DSP (predefined symbol)	26
__ARM_FEATURE_FMA (predefined symbol)	26
__ARM_FEATURE_IDIV (predefined symbol)	26
__ARM_FEATURE_NUMERIC_MAXMIN (predefined symbol)	26
__ARM_FP (predefined symbol)	27

__ARM_MEDIA__ (predefined symbol)	27	-U (assembler option)	61
__ARM_MPCORE__ (predefined symbol)	27	-w (assembler option)	61
__ARM_NEON (predefined symbol)	27	-x (assembler option)	62
__ARM_NEON_FP (predefined symbol)	27	--arm (assembler option)	46
__ARM_PROFILE_M__ (predefined symbol)	27	--cmse (assembler option)	47
__BUILD_NUMBER__ (predefined symbol)	28	--cpu (assembler option)	47
__CORE__ (predefined symbol)	28	--cpu_mode (assembler option)	48
__DATE__ (predefined symbol)	28	--endian (assembler option)	49
__FILE__ (predefined symbol)	28	--fpu (assembler option)	50
__IAR_SYSTEMS_ASM__ (predefined symbol)	28	--legacy (assembler option)	54
__IASMARM__ (predefined symbol)	28	--no_dwarf3_cfi (assembler option)	55
__LINE__ (predefined symbol)	28	--no_it_verification (assembler option)	55
__LITTLE_ENDIAN__ (predefined symbol)	28	--no_literal_pool (assembler option)	55
__TID__ (predefined symbol)	28	--no_path_in_file_macros (assembler option)	56
__TIME__ (predefined symbol)	28	--source_encoding (assembler option)	59
__VER__ (predefined symbol)	28	--suppress_vfe_header (assembler option)	59
- (assembler operator)	67–68	--system_include_dir (assembler option)	60
-B (assembler option)	46	--thumb (assembler option)	60
-c (assembler option)	47	:AND: (assembler operator)	70
-D (assembler option)	48	:EOR: (assembler operator)	71
-E (assembler option)	49	:LAND: (assembler operator)	70
-e (assembler option)	49	:LEOR: (assembler operator)	78
-f (assembler option)	44, 50	:LNOT: (assembler operator)	72
-G (assembler option)	51	:LOR: (assembler operator)	72
-g (assembler option)	51	:MOD: (assembler operator)	71
-I (assembler option)	51	:NOT: (assembler operator)	71
-i (assembler option)	52	:OR: (assembler operator)	71
-j (assembler option)	52, 141	:SHL: (assembler operator)	72
-L (assembler option)	53	:SHR: (assembler operator)	73
-l (assembler option)	53	! (assembler operator)	72
-M (assembler option)	54	!= (assembler operator)	69
-N (assembler option)	55	() (assembler operator)	66
-O (assembler option)	56	* (assembler operator)	66
-o (assembler option)	57	/ (assembler operator)	68
-p (assembler option)	57	/*...*/ (assembler directive)	118
-r (assembler option)	58	// (assembler directive)	118
-S (assembler option)	58	& (assembler operator)	70
-s (assembler option)	58	&& (assembler operator)	70
-t (assembler option)	60	#define (assembler directive)	111

<code>#elif</code> (assembler directive)	111
<code>#else</code> (assembler directive)	111
<code>#endif</code> (assembler directive)	111
<code>#error</code> (assembler directive)	111
<code>#if</code> (assembler directive)	111
<code>#ifdef</code> (assembler directive)	111
<code>#ifndef</code> (assembler directive)	111
<code>#include</code> files	52
<code>#include</code> files, specifying	51
<code>#include</code> (assembler directive)	111
<code>#message</code> (assembler directive)	111
<code>#pragma</code> (assembler directive)	111
<code>#undef</code> (assembler directive)	111
<code>^</code> (assembler operator)	71
<code>+</code> (assembler operator)	67
<code><</code> (assembler operator)	68
<code><<</code> (assembler operator)	72
<code><=</code> (assembler operator)	68
<code><></code> (assembler operator)	69
<code>=</code> (assembler directive)	94
<code>=</code> (assembler operator)	69
<code>==</code> (assembler operator)	69
<code>></code> (assembler operator)	69
<code>>=</code> (assembler operator)	70
<code>>></code> (assembler operator)	73
<code> </code> (assembler operator)	71
<code> </code> (assembler operator)	72
<code>~</code> (assembler operator)	71
<code>\$</code> (assembler directive)	118
<code>\$</code> (program location counter)	24

Numerics

32-bit expressions, loading in register	133
---	-----