IAR Assembler User Guide

for Arm Limited's Arm© cores

May 2025
AARM-15

Table of Contents

Preface **8**
Who should read this guide 8
How to use this guide 8
What this guide contains 8
Other documentation 9
User and reference guides 9
Document conventions 9
Typographic conventions 9
Naming conventions 10
Introduction to the IAR Assembler for Arm **12**
Introduction to assembler programming 12
Getting started 13
Modular programming 13
External interface details 14
Assembler invocation syntax 14
Passing options 14
Environment variables 14
Error return codes 15
Source format 15
Assembler instructions 15
Execution modes 16
Expressions, operands, and operators 16
Integer constants 16
ASCII character constants 17
Floating-point constants 17
True and false 18
Symbols 18
Labels 18
Register symbols 18
Predefined symbols 19
Absolute and relocatable expressions 23
Expression restrictions 23
List file format 24
Header 24
Body 24
Summary 24
Symbol and cross-reference table 24
Programming hints 25
Accessing special function registers 25
Using C-style preprocessor directives 25
Tracking call frame usage 25
Call frame information overview 26
Call frame information in more detail 26
Defining a names block 26
Defining a common block 27
Annotating your source code within a data block 28
Specifying rules for tracking resources and the stack depth 28
Using CFI expressions for tracking complex cases 30
Stack usage analysis directives 30
Examples of using CFI directives 31
Assembler options **33**
Using command line assembler options 34

Specifying options and their parameters 34
Extended command line file 35
Summary of assembler options 35
Descriptions of assembler options 36
--aarch32 36
--aarch64 36
--abi 37
--arm 37
-B 38
-c 38
--cmse 38
--cpu 39
--cpu_mode 39
-D 40
--diagnostics_format 40
--dynamic_output 41
-E 41
-e 41
--enable_hardware_workaround 42
--endian 42
-f 42
--f 43
--fpu 43
-G 44
-g 44
-I 44
-i 45
-j 45
-L 46
-l 46
--legacy 47
-M 47
-N 48
--no_dwarf3_cfi 48
--no_dwarf4 48
--no_it_verification 48
--no_literal_pool 49
--no_path_in_file_macros 49
-O 49
-o 50
-p 50
-r 51
-S 51
-s 51
--source_encoding 52
--suppress_vfe_header 52
--system_include_dir 52
-t 53
--thumb 53
-U 53
--version 54
-w 54
-x 55
-Y 55
-y 56

Assembler operators **57**
Precedence of assembler operators 58
Summary of assembler operators 58
Parenthesis operator 58
Unary operators 58
Multiplicative arithmetic operators 59
Additive arithmetic operators 59
Shift operators 59
AND operators 59
OR operators 59
Comparison operators 59
Description of assembler operators 60
() Parenthesis 60
* Multiplication 60
+ Unary plus 60
+ Addition 61
– Unary minus 61
– Subtraction 61
/ Division 61
< Less than 62
<= Less than or equal to 62
<>, != Not equal to 62
=, == Equal to 62
> Greater than 63
>= Greater than or equal to 63
&& Logical AND 63
& Bitwise AND 64
~ Bitwise NOT 64
| Bitwise OR 64
^ Bitwise exclusive OR 64
% Modulo 65
! Logical NOT 65
|| Logical OR 65
<< Logical shift left 65
>> Logical shift right 66
BYTE1 First byte 66
BYTE2 Second byte 66
BYTE3 Third byte 67
BYTE4 Fourth byte 67
DATE Current time/date 67
HIGH High byte 67
HWRD High word 68
LOW Low byte 68
LO12 Lower 12 bits of symbol 68
LWRD Low word 68
SBREL 69
SFB section begin 69
SFE section end 70
SIZEOF section size 70
UGT Unsigned greater than 71
ULT Unsigned less than 71
XOR Logical exclusive OR 71
Assembler directives **73**
Summary of assembler directives 73
Description of assembler directives 76

Module control directives ... 76
Symbol control directives ... 78
Mode control directives ... 80
Section control directives ... 81
Value assignment directives ... 84
Conditional assembly directives ... 86
Macro processing directives ... 87
Listing control directives ... 95
C-style preprocessor directives ... 98
Data definition or allocation directives ... 102
Assembler control directives ... 105
Function directives ... 108
Call frame information directives for names blocks ... 108
Call frame information directives for common blocks ... 109
Call frame information directives for data blocks ... 110
Call frame information directives for tracking resources and CFAs ... 111
Call frame information directives for stack usage analysis ... 113

Assembler pseudo-instructions ... 115

Summary ... 115
Descriptions of pseudo-instructions ... 116
ADR (ARM) ... 116
ADR (CODE16) ... 117
ADR (THUMB) ... 117
ADRL (64-bit mode) ... 118
ADRL (ARM) ... 118
ADRL (THUMB) ... 119
LDR (64-bit mode) ... 119
LDR (ARM) ... 119
LDR (CODE16) ... 120
LDR (THUMB) ... 120
MOV (CODE16) ... 121
MOV32 (THUMB) ... 122
MOVL (64-bit mode) ... 122
NOP (ARM) ... 123
NOP (CODE16) ... 123

Assembler diagnostics ... 124

Message format ... 124
Severity levels ... 124
Options for diagnostics ... 124
Assembler warning messages ... 124
Command line error messages ... 124
Assembler error messages ... 125
Assembler fatal error messages ... 125
Assembler internal error messages ... 125

Migrating to the IAR Assembler for Arm ... 126

Introduction ... 126
Thumb code labels ... 126
Alternative register names ... 127
Alternative mnemonics ... 127
Operator synonyms ... 128
Warning messages ... 129
The first register operand omitted ... 129
The first register operand duplicated ... 129
Immediate #0 omitted in Load/Store ... 129

List of Tables

1. Typographic conventions used in IAR documentation 10
2. Naming conventions used in IAR documentation 10
3. Assembler environment variables 14
4. Assembler error return codes 15
5. Assembler source line components 15
6. Integer constant formats 17
7. ASCII character constant formats 17
8. Floating-point constants 17
9. Predefined register symbols in 32-bit mode 18
10. Predefined 64-bit register symbols in 64-bit mode 19
11. Predefined symbols 19
12. Symbol and cross-reference table 24
13. Code sample with backtrace rows and columns 31
14. Assembler options summary 35
15. Assembler directives summary 74
16. Module control directives 77
17. Symbol control directives 79
18. Mode control directives 80
19. Section control directives 82
20. Value assignment directives 85
21. Macro processing directives 87
22. Listing control directives 95
23. C-style preprocessor directives 99
24. Data definition or allocation directives 103
25. Assembler control directives 105
26. Call frame information directives names block 109
27. Call frame information directives common block 110
28. Call frame information directives for data blocks 111
29. Unary operators in CFI expressions 112
30. Binary operators in CFI expressions 112
31. Ternary operators in CFI expressions 112
32. Call frame information directives for tracking resources and CFAs 113
33. Call frame information directives for stack usage analysis 114
34. Pseudo-instructions for A32 and T32 115
35. Pseudo-instructions for A64 116
36. Alternative register names 127
37. Alternative mnemonics 127
38. Operator synonyms 128

Preface

Contents

Who should read this guide 8
How to use this guide 8
What this guide contains 8
Other documentation 9
User and reference guides 9
Document conventions 9
Typographic conventions 9
Naming conventions 10

Welcome to the *IAR Assembler User Guide for Arm*—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, page 12*.

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 IAR Embedded Workbench, we suggest that you first go through the tutorials, which you can find in IAR Information Center in the product, under **Product Explorer**. They will help you get started.

WHAT THIS GUIDE CONTAINS

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

- *Introduction to the IAR Assembler for Arm, page 12* provides programming information. It also describes the source code format, and the format of assembler listings.
- *Assembler options, page 33* 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, page 57* gives a summary of the assembler operators, arranged in order of precedence, and provides detailed reference information about each operator.

- *Assembler directives, page 73* 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, page 115* lists the pseudo-instructions that are accepted by the IAR Assembler for Arm.
- *Assembler diagnostics, page 124* contains information about the formats and severity levels of diagnostic messages.
- *Migrating to the IAR Assembler for Arm, page 126* contains information that is useful when migrating from an existing product to the IAR Assembler for Arm.

OTHER DOCUMENTATION

User documentation is available as hypertext PDFs and as a help system in HTML format. You can access the documentation from the IAR Information Center or from the **Help** menu in the IAR Embedded Workbench IDE.

User and reference guides

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

- System requirements and information about how to install and register the IAR products are available in the *Installation and Licensing Quick Reference Guide* and the *Licensing Guide*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide for Arm.*
- Using the IAR C-SPY® Debugger and C-RUN runtime error checking, is available in the *C-SPY Debugging Guide for Arm.*
- Programming for the IAR C/C++ Compiler for Arm and linking, is available in the *IAR C/C++ Development Guide for Arm.*
- Programming for the IAR Assembler for Arm is available in the *IAR Assembler User Guide for Arm.*
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT® Static Analysis Guide*.
- Using I-jet, refer to the *IAR Debug Probes User Guide for I-jet®, I-jet Trace, and I-scope*.
- Using J-Link and J-Trace, refer to the J-Link/J-Trace documentation available at *www.segger.com*.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for Arm, is available in the *IAR Embedded Workbench® Migration Guide*.

Additional documentation might be available depending on your product installation.

DOCUMENT CONVENTIONS

When, in the IAR 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:\iar\ewarm-2.1\arm\doc`.

Typographic conventions

The IAR documentation set uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file.
`[option]`	An optional part of a linker or stack usage control 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 linker or stack usage control 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 line option, pragma directive, or library filename.
`[a|b|c]`	An optional part of a command line option, pragma directive, or library filename with alternatives.
`{a|b|c}`	A mandatory part of a command line option, pragma directive, or library filename with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• 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 IAR documentation

Naming conventions

The following naming conventions are used for the products and tools from IAR, 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
IAR ILINK Linker™	ILINK, the linker
IAR DLIB Runtime Environment™	the DLIB runtime environment

Table 2. Naming conventions used in IAR documentation

In 32-bit mode refers to using IAR Embedded Workbench for Arm configured for the instruction sets T32/T and A32.

In 64-bit mode refers to using IAR Embedded Workbench for Arm configured for the instruction set A64.

For more information, see *Execution modes, page 16*.

Introduction to the IAR Assembler for Arm

Contents

Introduction to assembler programming 12
Getting started 13
Modular programming 13
External interface details 14
Assembler invocation syntax 14
Passing options 14
Environment variables 14
Error return codes 15
Source format 15
Assembler instructions 15
Execution modes 16
Expressions, operands, and operators 16
Integer constants 16
ASCII character constants 17
Floating-point constants 17
True and false 18
Symbols 18
Labels 18
Register symbols 18
Predefined symbols 19
Absolute and relocatable expressions 23
Expression restrictions 23
List file format 24
Header 24
Body 24
Summary 24
Symbol and cross-reference table 24
Programming hints 25
Accessing special function registers 25
Using C-style preprocessor directives 25
Tracking call frame usage 25
Call frame information overview 26
Call frame information in more detail 26
Defining a names block 26
Defining a common block 27
Annotating your source code within a data block 28
Specifying rules for tracking resources and the stack depth 28
Using CFI expressions for tracking complex cases 30
Stack usage analysis directives 30
Examples of using CFI directives 31

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, under Product Explorer
- 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 enable you to 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

You can use the assembler either from the IAR Embedded Workbench IDE or from the command line. Refer to the *IDE Project Management and Building Guide for Arm* for information about using the assembler from the IAR Embedded Workbench IDE.

Assembler invocation syntax

The invocation syntax for the assembler is:

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

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

```
iasmarm prog -r
```

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

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

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

Passing options

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

- Directly from the command line
  Specify the options on the command line after the `iasmarm` command, see *Assembler invocation syntax, page 14*.
- 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 14*.
- Via a text file by using the `-f` option, see *-f, page 42*.

For general guidelines for the option syntax, an options summary, and more information about each option, see *Assembler options, page 33*.

Environment variables

You can use these environment variables with the IAR Assembler:

Environment variable	Description
`IASMARM`	Specifies command line options, for example: `set IASMARM=-L -ws`
`IASMARM_INC`	Specifies directories to search for include files, for example: `set IASMARM_INC=c:\myinc\`

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 `-ws` option is used).
`2`	Errors occurred.
`3`	Fatal errors occurred (making the assembler abort).

Table 4. Assembler error return codes

SOURCE FORMAT

The format of an assembler source line is as follows:

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

where the components are as follows:

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

Table 5. Assembler source line components

The components are separated by spaces or tabs.

A source line cannot exceed 2,047 characters.

Tab characters, ASCII `09H`, are expanded according to the most common practice, that is, 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.

EXECUTION MODES

IAR Embedded Workbench for Arm supports the 32-bit and 64-bit Arm architectures by means of execution modes.

In 32-bit mode refers to using IAR Embedded Workbench for Arm configured to generate and debug code for the T32/T and A32 instruction sets, either on an Armv4/5/6/7 core or in the AArch32 execution state on an Arm v8-A core. In 32-bit mode, you can use both the A32 and T32/T instruction sets and switch between them.

In 64-bit mode refers to using IAR Embedded Workbench for Arm configured to generate and debug code for the A64 instruction set in the AArch64 execution state on an Arm v8-A core. Code in 64-bit mode can call code in 32-bit mode, and that code can return back. However, the IAR translator tools do not support this switch being used in a single linked image. Switching between A32/T32/T code and A64 code must be performed by using several images. For example, an OS using 64-bit mode can start applications in either 64-bit or in 32-bit mode.

The AArch32 execution state is compatible with the Arm v7 architecture. The AArch32 execution state is emulated inside the AArch64 execution state.

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 64-bit two’s complement integers. Range checking is performed if a value is used for generating code.

Expressions are evaluated from left to right, unless this order is overridden by the priority of operators. See also *Assembler operators, page 57*.

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.

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

The assembler uses 64-bit two’s complement internal arithmetic, so integers have a (signed) range from $-2^{63}-1$ to $2^{63}-1$.

Constants are written as a sequence of digits with an optional preceding `-` (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 6. Integer constant formats

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 7. ASCII character constant formats

Floating-point constants

The IAR Assembler accepts floating-point values as constants and converts them into IEEE half-precision (16-bit), single-precision (32-bit) or double-precision (64-bit) floating-point format, or fractional format.

Floating-point numbers can be written in the format:

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

This table shows valid examples:

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

Table 8. Floating-point constants

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

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 32,000 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 51*.

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 102*.

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*.

To refer to the program location counter in your assembler source code, use the **.** (period) character. For example:

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

Register symbols

This table shows the predefined register symbols available in 32-bit mode:

Name	Size	Description
`CPSR`	32 bits	Current program status register
`D0-D31`	64 bits	Floating-point coprocessor registers for double precision
`FPCXT`	32 bits	Floating-point context payload
`FPEXC`	32 bits	Floating-point coprocessor, exception register

Name	Size	Description
`FPSCR`	32 bits	Floating-point coprocessor, status and control register
`FPSID`	32 bits	Floating-point coprocessor, system ID register
`Q0-Q15`	128 bits	Advanced `SIMD` registers
`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 9. Predefined register symbols in 32-bit mode

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

This table shows the predefined register symbols available in 64-bit mode:

Name	Size	Description
`X0-X30`	64 bits	64 bits in 64-bits general purpose registers `R0-R30`
`W0-W30`	32 bits	32 bits in 64-bits general purpose registers `R0-R30`
`SP`	64 bits	Stack pointer
`WSP`	32 bits	Stack pointer
`V0-V31`	128 bits	128-bit SIMD and floating-point registers `V0–V31`
`Q0-Q31`	128 bits	128-bit entity in 128-bit SIMD and floating-point registers `V0–V31`
`D0-D31`	64 bits	Double-precision floating-point in 128-bit SIMD and floating-point registers `V0–V31`
`S0-S31`	32 bits	Single-precision floating-point in 128-bit SIMD and floating-point registers `V0–V31`
`H0-H31`	16 bits	Half-precision floating-point in 128-bit SIMD and floating-point registers `V0–V31`
`B0-B31`	8 bits	8-bit entity in 128-bit SIMD and floating-point registers `V0–V31`
`IP0`	64 bits	First intra-procedure-call scratch register, alias to `R16`
`IP1`	64 bits	Second intra-procedure-call scratch register, alias to `R17`
`FP`	64 bits	Frame pointer, alias to `R29`
`LR`	64 bits	Link register, alias to `R30`
`XZR`	64 bits	Zero 64-bit register
`WZR`	32 bits	Zero 32-bit register

Table 10. Predefined 64-bit register symbols in 64-bit mode

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
`__aarch64__`	This symbol is defined to `1` when assembling for the A64 instruction set in the AArch64 state.

Symbol	Value
`__ARM_32BIT_STATE`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_64BIT_STATE`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_ADVANCED_SIMD__`	An integer that is set based on the `--cpu` option. The symbol is set to `1` if the selected processor architecture has the Advanced SIMD architecture extension. The symbol is undefined for other cores.
`__ARM_ARCH`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_ARCH_ISA_A64`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_ARCH_ISA_ARM`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_ARCH_ISA_THUMB`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_ARCH_PROFILE`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_BIG_ENDIAN`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_AES`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_CLZ`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_CMSE`	An integer that is set based on the assembler option `--cpu` and `--cmse`. The symbol is set to `3` if the selected processor architecture has CMSE (Cortex-M security extensions) and the assembler option `--cmse` is specified. The symbol is set to `1` if the selected processor architecture has CMSE and the assembler option `--cmse` is not specified. The symbol is undefined for cores without CMSE.
`__ARM_FEATURE_CRC32`	This symbol is defined to `1` if the CRC32 instructions are supported (optional in Armv8-A/R). This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_CRYPTO`	This symbol is defined to `1` if the cryptographic instructions are supported (implies Armv8-A/R with Neon). This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_DIRECTED_ROUNDING`	This symbol is defined to `1` if the directed rounding and conversion instructions are supported. This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_DSP`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_FMA`	This symbol is defined to `1` if the FPU supports fused floating-point multiply-accumulate.

Symbol	Value
	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_FP16_FML`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_IDIV`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_NUMERIC_MAXMIN`	This symbol is defined to `1` if the floating-point numeric maximum and minimum instructions are supported. This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_QBIT`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_QRDMX`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_SAT`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_SHA2`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_SHA3`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_SHA512`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_SIMD32`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_SM3`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FEATURE_SM4`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_FP`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_MEDIA__`	An integer that is set based on the `--cpu` option. The symbol is set to `1` if the selected processor architecture has the ARMv6 SIMD extension for multimedia. The symbol is undefined for other cores.
`__ARM_MPCORE__`	An integer that is set based on the `--cpu` option. The symbol is set to `1` if the selected processor architecture has the Multiprocessing Extensions. The symbol is undefined for other cores.
`__ARM_NEON`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_NEON_FP`	This symbol is defined according to the *Arm C Language Extensions* (ACLE).
`__ARM_PROFILE_M__`	An integer that is set based on the `--cpu` option. The symbol is set to `1` if the selected processor is a profile M core. The symbol is undefined for other cores.
`__ARMVFP__`	An integer that is set based on the `--fpu` option and that identifies whether floating-point instructions for a vector floating-point coprocessor have been enabled or not. The symbol is defined to `__ARMVFPV2__`, `__ARMVFPV3__`, or `__ARMVFPV4__`. These symbolic names can be used when

Symbol	Value
	testing the `__ARMVFP__` symbol. If floating-point instructions are disabled (default), the symbol is undefined.
`__ARMVFP_D16__`	An integer that is set based on the assembler option `--fpu`. 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.
`__ARMVFP_FP16__`	An integer that is set based on the assembler option `--fpu`. The symbol is set to `1` if the selected FPU only supports 16-bit floating-point numbers. Otherwise, the symbol is undefined.
`__ARMVFP_SP__`	An integer that is set based on the assembler option `--fpu`. The symbol is set to `1` if the selected FPU only supports 32-bit single-precision. Otherwise, the symbol is undefined.
`__BUILD_NUMBER__`	A unique integer that identifies the build number of the assembler currently in use. The build number does not necessarily increase with an assembler that is released later.
`__CORE__`	An integer that identifies the chip core in use. The value reflects the setting of the assembler option `--cpu`. For information about the possible values, see the *IAR C/C++ Development Guide for Arm*.
`__DATE__`	The current date in `dd/Mmm/yyyy` format (string).
`__FILE__`	The name of the current source file (string).
`__IAR_SYSTEMS_ASM__`	IAR assembler identifier (number). Note that the number could be higher in a future version of the product. This symbol can be tested with `#ifdef` to detect whether the code was assembled by an assembler from IAR.
`__IASMARM__`	An integer that is set to `1` when the code is assembled with the IAR Assembler for Arm.
`__ilp32__`	This symbol is defined to `1` when assembling for the A64 instruction set in the AArch64 state, using the ILP32 data model.
`__LINE__`	The current source line number (number).
`__LITTLE_ENDIAN__`	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.
`__lp64__`	This symbol is defined to `1` when assembling for the A64 instruction set in the AArch64 state, using the LP64 data model.
`__TID__`	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.
`__TIME__`	The current time in `hh:mm:ss` format (string).
`__VER__`	The version number in integer format, for example, version 6.21.2 is returned as 6021002 (number).

Table 11. Predefined symbols

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

Testing symbols for conditional assembly

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

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 86*.

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 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       dc32    5                ; A relocatable label.
second      equ     10 + 5           ; An absolute expression.

            dc32    first            ; Examples of some legal
            dc32    first + 1        ; relocatable expressions.
            dc32    first + second
            end
```

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

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 12. Symbol and cross-reference table

PROGRAMMING HINTS

Accessing special function registers

Specific header files for several Arm devices are included in the IAR 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 105*.

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 26*
- *Call frame information in more detail, page 26*

These tasks are described:

- *Defining a names block, page 26*
- *Defining a common block, page 27*
- *Annotating your source code within a data block, page 28*
- *Specifying rules for tracking resources and the stack depth, page 28*
- *Using CFI expressions for tracking complex cases, page 30*
- *Stack usage analysis directives, page 30*
- *Examples of using CFI directives, page 31*

For reference information, see:

- *Call frame information directives for names blocks, page 108*
- *Call frame information directives for common blocks, page 109*
- *Call frame information directives for data blocks, page 110*
- *Call frame information directives for tracking resources and CFAs, page 111*
- *Call frame information directives for stack usage analysis, page 113*

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. This information is used, for example, in the **Call Stack** window.
- 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 therefore 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 that corresponds to the device architecture, either *DWARF for the ARM architecture* or *DWARF for the Arm 64-bit architecture (AArch64).* 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 109*. For more information about how to use these directives, see *Specifying rules for tracking resources and the stack depth, page 28* and *Using CFI expressions for tracking complex cases, page 30*.

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 110*. For more information on how to use these directives, see *Specifying rules for tracking resources and the stack depth, page 28*, and *Using CFI expressions for tracking complex cases, page 30*.

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 30*. 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 nocalls
        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 111*.

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 Armcore. 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 13. 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
```

☞ `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     funcall func2

            bl      func2
            pop     {r0,lr}

            cfi     r1  r0
            cfi     r14 samevalue
            cfi     cfa r13

            mov     r1,r0


            cfi     r1 samevalue

            bx      lr

            cfi     endblock trivialBlock

            end
```

Assembler options

Contents

Using command line assembler options 34
Specifying options and their parameters 34
Extended command line file 35
Summary of assembler options 35
Descriptions of assembler options 36
--aarch32 36
--aarch64 36
--abi 37
--arm 37
-B 38
-c 38
--cmse 38
--cpu 39
--cpu_mode 39
-D 40
--diagnostics_format 40
--dynamic_output 41
-E 41
-e 41
--enable_hardware_workaround 42
--endian 42
-f 42
--f 43
--fpu 43
-G 44
-g 44
-I 44
-i 45
-j 45
-L 46
-l 46
--legacy 47
-M 47
-N 48
--no_dwarf3_cfi 48
--no_dwarf4 48
--no_it_verification 48
--no_literal_pool 49
--no_path_in_file_macros 49
-O 49
-o 50
-p 50
-r 51
-S 51
-s 51
--source_encoding 52
--suppress_vfe_header 52
--system_include_dir 52
-t 53
--thumb 53

-U 53
--version 54
-w 54
-x 55
-Y 55
-y 56

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 *IDE Project Management and Building 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.

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

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

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 options accept a parameter that is not a filename. For options with a long name, the option and the parameter can be separated with a space character, an = sign, or a `#`. For options with a short name, the parameter 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\
```

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

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.

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
`--aarch32`	Generates code using the A32 instruction set
`--aarch64`	Generates code using the A64 instruction set
`--abi`	Specifies a data model for generating code using the A64 instruction set
`--arm`	Sets the default mode for the assembler directive `CODE` to Arm
`-B`	Macro execution information
`-c`	Conditional list
`--code_model`	Enables CMSE secure object generation
`--cpu`	Core configuration
`--cpu_mode`	Sets the mode for the assembler directive `CODE`
`-D`	Defines preprocessor symbols
`--diagnostics_format`	Specifies the format for printed diagnostics
`--dynamic_output`	Lists in a structured format all output files
`-E`	Maximum number of errors
`-e`	Generates code in big-endian byte order
`--enable_hardware_workaround`	Enables a specific hardware workaround
`--endian`	Specifies the byte order for code and data
`-f`	Extends the command line
`--fpu`	Floating-point coprocessor architecture configuration
`-G`	Opens standard input as source
`-g`	Disables the automatic search for system include files
`-I`	Adds a search path for a header file
`-i`	Lists #included text
`-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_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 `__FILE__` and `__BASE_FILE__`
`-O`	Sets the object filename to path

Command line option	Description
`-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` to Thumb
`-U`	Undefines a symbol
`--use_unix_directory_ separators`	Uses `/` as directory separator in paths
`--version`	Sends assembler output to the console and then exits.
`-w`	Disables warnings
`-x`	Includes cross-references
`-Y`	Generates a list of file dependencies to a path
`-y`	Generates a list of file dependencies

Table 14. Assembler options summary

DESCRIPTIONS OF ASSEMBLER OPTIONS

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

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.

--aarch32

Syntax

```
--aarch32
```

Description

Use this option to generate code using the A32 instruction set in the AArch32 state for the assembler directive `CODE`.

This option has the same effect as the `--cpu_mode=arm` option.

See also

--cpu_mode, page 39.

To set this option, use **Project>Options>General Options>Target>Execution mode**

--aarch64

Syntax

```
--aarch64
```

Description

Use this option to generate code using the A64 instruction set in the AArch64 state for the assembler directive `CODE`.

This option has the same effect as the `--cpu_mode=a64` option.

See also

--abi, page 37 and *--cpu_mode, page 39*.

To set this option, use **Project>Options>General Options>Target>Execution mode**

--abi

Syntax

```
--abi={ilp32|lp64}
```

Parameters

`ilp32`	Generates A64 code for the ILP32 data model. Defines the symbol `__ilp32__`
`lp64`	Generates A64 code for the LP64 data model. Defines the symbol `__lp64__`.

Description

Use this option to specify a data model for the generation of code using the A64 instruction set in the AArch64 environment.

See also

--aarch64, page 36 and *--cpu_mode, page 39*.

To set related options, choose:

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

and

Project>Options>General Options>64-bit>Data model

--arm

Syntax

```
--arm
```

Description

Use this option to make Arm (A32 **in 32-bit mode**) the default mode for the assembler directive `CODE`.

See also

--cpu_mode, page 39

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:

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

This option is mainly used in conjunction with the list file options `-L` or `-l`.

See also

-L, page 46.

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 46.

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`. **In 64-bit mode**, this option has no effect.

To use this option, you must first select the option **Project>Options>General Options>32-bit>TrustZone**.

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` or `8-a.AArch64`. The default value **in 32-bit mode** is `ARM7TDMI` and **in 64-bit mode** it is `Cortex-A53`.

Description

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

See also

The reference documentation for the compiler option `--cpu` in the *IAR C/C++ Development Guide for Arm* for a list of supported architectures and processor variants.

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

--cpu_mode

Syntax

```
--cpu_mode {arm|a|thumb|t|aarch64|a64}
```

Parameters

`aarch64` or `a64`	Uses the A64 instruction set in the AArch64 state
`arm` or `a`	Uses the A32 instruction set in 32-bit mode
`thumb` or `t`	Uses the T32 or T instruction set in 32-bit mode

Description

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

See also

--aarch64, page 36

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

-D

Syntax

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

Parameters

symbol	The name of the symbol you want to define.
value	The value of the symbol. If no value is specified, `1` is used.

Description

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

Example

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

```
#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

--diagnostics_format

Syntax

```
--diagnostics_format format
```

Parameters

format can be one of:

`text`	Console output. This is the default if no `--diagnostics_format` option is given on the command line.
`sarif_stderr`	Output in SARIF Version 2.1.0 format to `stderr`.
`sarif_stdout`	Output in SARIF Version 2.1.0 format to `stdout`.
`sarif_file`	Output in SARIF Version 2.1.0 format to a `.sarif` file named after the source file.

Description

Use this option to choose the format of the printed diagnostics.

This option is not available in the IDE.

--dynamic_output

Syntax

```
--dynamic_output filename
```

Parameters

Description

Use this option to make the assembler list in a structured format the names of all output files from the assembly, except for the generated dynamic output file itself.

This option is not available in the IDE.

-E

Syntax

```
-Enumber
```

Parameters

number	The number of errors before the assembler stops the assembly. `number` must be a positive integer —`0` indicates no limit.

Description

Use this option to specify the maximum number of errors that the assembler reports. By default, the maximum number is 100.

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

-e

Syntax

```
-e
```

Description

Use this option to cause the assembler to generate code and data in big-endian byte order. The default byte order is little-endian.

In 64-bit mode, this option has no effect.

Project>Options>General Options>32-bit>Byte order

--enable_hardware_workaround

Syntax

```
--enable_hardware_workaround=waid[,waid...]
```

Parameters

waid	The ID number of the workaround to enable. For a list of available workarounds to enable, see the release notes.

Description

Use this option to make the assembler generate a workaround for a specific hardware problem.

See also

The release notes for the assembler for a list of available parameters.

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

--endian

Syntax

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

Parameters

`little`, `l` (default)	Specifies little-endian byte order.
`big`, `b`	Specifies big-endian byte order.

Description

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

In 64-bit mode, this option has no effect.

Project>Options>General Options>32-bit>Byte order

-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 34*.

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

-f, page 42 and *Extended command line file, page 35*.

To set this option, use:

Project>Options>Assembler>Extra Options

--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 *Specifying options and their parameters, page 34*.

Description

Use this option to make the assembler read command line options from the named file, with the default filename extension `xcl`.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character acts just as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

If you use the assembler option `-Y`, extended command line files specified using `--f` will generate a dependency, but those specified using `-f` will not generate a dependency.

See also

-Y, page 55 and *-f, page 42*.

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.

In 64-bit mode, and when a 64-bit device is used in 32-bit mode, this option has no effect.

See also

The reference documentation for the compiler option `--cpu` in the *IAR C/C++ Development Guide for Arm* for a list of supported coprocessor architecture variants.

Project>Options>General Options>32-bit>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 128 and the chapter *Migrating to the IAR Assembler for Arm, page 126*.

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

-L

Syntax

```
-L[path]
```

Parameters

No parameter	Generates a listing with the same name as the source file, but with the filename extension `lst`.
path	The path to the destination of the list file. Note that you must not include a space before the path.

Description

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

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

Example

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

```
iasmarm prog -Llist\
```

To set related options, select:

Project>Options>Assembler >List

-l

Syntax

```
-l {filename|directory}
```

Parameters

filename	The output is stored in the specified file. Note that you must include a space before the filename. If no extension is specified, `lst` is used.
directory	The output is stored in a file (filename extension `lst`) which is stored in the specified directory.

For information about specifying a filename or directory, see *Using command line assembler options, page 34*.

Description

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

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

To set related options, select:

Project>Options>Assembler >List

--legacy

Syntax

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

Parameters

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

Description

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

In 64-bit mode, this option has no effect.

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

-M

Syntax

```
-Mab
```

Parameters

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

Description

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

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

Example

For example, using the option:

```
-M[]
```

in the source you would write, for example:

```
print [>]
```

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

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

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

Project>Options>Assembler >Language>Macro quote characters

-N

Syntax

```
-N
```

Description

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

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

See also

-L, page 46.

Project>Options>Assembler >List>Include header

--no_dwarf3_cfi

Syntax

```
--no_dwarf3_cfi
```

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 a does not support DWARF 3.

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

--no_dwarf4

Syntax

```
--no_dwarf4
```

Description

Use this option to suppress generation of DWARF 4 debug information. This can lead to a degraded debugging experience, but might allow loading in a debugger that a does not support DWARF 4.

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

--no_it_verification

Syntax

```
--no_it_verification
```

Description

Use this option to suppress the verification of the condition of instructions following an `IT` instruction.

In 64-bit mode, this option has no effect.

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

--no_literal_pool

Syntax

```
--no_literal_pool
```

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 Armv6-M and Armv7 compatible cores (includes Armv8-M, Armv8.1-M, Armv8-A and Armv8-R cores).

In 64-bit mode, this option has no effect.

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 cannot 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.

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

Example

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

```
iasmarm prog -Oobj\
```

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

-o

Syntax

```
-o {filename|directory}
```

Parameters

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

For information about specifying a filename or directory, see *Using command line assembler options, page 34*.

Description

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

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

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

-p

Syntax

```
-plines
```

Parameters

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

Description

Use this option to set the number of lines per page explicitly.

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

See also

-L, page 46

-l, page 46.

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 the 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

```
--system_include_dir path
```

Parameters

`path`	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

```
-tn
```

Parameters

`n`	The tab spacing—must be in the range `2` to `9`.

Description

By default, the assembler sets 8 character positions per tab stop. Use this option to specify a different tab spacing.

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

See also

-L, page 46

-l, page 46.

Project>Options>Assembler>List>Tab spacing

--thumb

Syntax

```
--thumb
```

Description

Use this option to make Thumb (T32 or T **in 32-bit mode**) the default mode for the assembler directive `CODE`.

See also

--cpu_mode, page 39

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

-U

Syntax

```
-Usymbol
```

Parameters

`symbol`	The predefined symbol to be undefined.

Description

By default, the assembler provides certain predefined symbols.

Use this option to undefine such a predefined symbol to make its name available for your own use through a subsequent `-D` option or source definition.

Example

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

```
iasmarm prog -U__TIME__
```

See also

Predefined symbols, page 19.

This option is not available in the IDE.

--version

Syntax

```
--version
```

Description

Use this option to make the assembler send version information to the console and then exit.

This option is not available in the IDE.

-w

Syntax

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

Parameters

No parameter	Disables all warnings.
`+`	Enables all warnings.
`-`	Disables all warnings.
`+n`	Enables just warning *n*.
`-n`	Disables just warning *n*.
`+m-n`	Enables warnings *m* to *n*.
`-m-n`	Disables warnings *m* to *n*.
`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.

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 124.

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 46

-l, page 46.

Project>Options>Assembler>List>Include cross reference

-Y

Syntax

```
-Y[path]
```

Parameters

path	The path to the location of the output file that contains the list of dependency files. Note that you cannot include a space before the path.

Description

Use this option to list each source file opened by the assembler in a file.

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

`-Y` cannot be used at the same time as `-y`.

Example

If `-y` or `-Y` is used, the output file will have one make rule. The object file will be dependent on all opened input files, including the full path. For example:

```
objectfile: sourcefile \
             inputfile \
```

This option is not available in the IDE.

-y

Syntax

```
-y {filename|directory}
```

Parameters

filename	The list of dependency files is stored in the specified file.
directory	The list of dependency files is stored in a file (filename extension `d`) which is stored in the specified directory.

For information about specifying a filename or directory, see *Using command line assembler options, page 34*.

Description

Use this option to list each source file opened by the assembler in a file. By default, the list is located in a file with the same name as the source file, but with the extension `d`. Use this option to specify a different output filename for the file.

The `-y` option cannot be used at the same time as the `-Y` option.

Example

If `-y` or `-Y` is used, the output file will have one make rule. The object file will be dependent on all opened input files, including the full path. For example:

```
objectfile: sourcefile \
             inputfile \
```

This option is not available in the IDE.

Assembler operators

Contents

Precedence of assembler operators 58
Summary of assembler operators 58
Parenthesis operator 58
Unary operators 58
Multiplicative arithmetic operators 59
Additive arithmetic operators 59
Shift operators 59
AND operators 59
OR operators 59
Comparison operators 59
Description of assembler operators 60
() Parenthesis 60
* Multiplication 60
+ Unary plus 60
+ Addition 61
– Unary minus 61
– Subtraction 61
/ Division 61
< Less than 62
<= Less than or equal to 62
<>, != Not equal to 62
=, == Equal to 62
> Greater than 63
>= Greater than or equal to 63
&& Logical AND 63
& Bitwise AND 64
~ Bitwise NOT 64
| Bitwise OR 64
^ Bitwise exclusive OR 64
% Modulo 65
! Logical NOT 65
|| Logical OR 65
<< Logical shift left 65
>> Logical shift right 66
BYTE1 First byte 66
BYTE2 Second byte 66
BYTE3 Third byte 67
BYTE4 Fourth byte 67
DATE Current time/date 67
HIGH High byte 67
HWRD High word 68
LOW Low byte 68
LO12 Lower 12 bits of symbol 68
LWRD Low word 68
SBREL 69
SFB section begin 69
SFE section end 70
SIZEOF section size 70
UGT Unsigned greater than 71
ULT Unsigned less than 71

XOR Logical exclusive OR 71

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.

The operator synonyms are enabled by the option `-j`. See also the chapter *Migrating to the IAR Assembler for Arm, page 126*.

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
`LO12`	Lower 12 bits of a symbol
`LWRD`	Low word
`HWRD`	High word
`DATE`	Current time/date

`SBREL`	The offset to a symbol from the addressing origin of its output segment.
`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 to
`<>`, `!=`	Not equal to
`>`	Greater than
`<`	Less than
`UGT`	Unsigned greater than
`ULT`	Unsigned less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

DESCRIPTION OF ASSEMBLER OPERATORS

This section gives detailed descriptions of each assembler operator.

See also *Expressions, operands, and operators, page 16*.

() 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 -> 4
-2*2 -> -4
```

+ Unary plus

Precedence

1

Description

Unary plus operator; performs nothing.

Example

```
+3 -> 3
3*+2 -> 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 -> 111
-2+2 -> 0
-2+-2 -> -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 -> -3
3*-2 -> -6
4--5 -> 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

```
92-19 -> 73
-2-2 -> -4
-2--2 -> 0
```

/ 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

```
9/2 -> 4
-12/3 -> -4
9/2*6 -> 24
```

< Less than

Precedence

7

Description

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

Example

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

<= Less than or equal to

Precedence

7

Description

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

Example

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

<>, != Not equal to

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 to

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 to

Precedence

7

Description

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

Example

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

&& 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

```
1010B && 0011B -> 1
1010B && 0101B -> 1
1010B && 0000B -> 0
```

& 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

```
1010B & 0011B -> 0010B
1010B & 0101B -> 0000B
1010B & 0000B -> 0000B
```

~ 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 -> 11111111111111111111111111110101B
```

| 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.

The precedence of `:SHL:` is 2.5.

Example

```
00011100B << 3 -> 11100000B
00000111111111111B << 5 -> 11111111111100000B
14 << 1 -> 28
```

>> Logical shift right

Precedence

4

Description

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

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
```

LO12 Lower 12 bits of symbol

Precedence

1

Description

LOW uses the lower 12 bits of a symbol. This operator is only available in 64-bit mode.

Example

```
LO12(0x1234) -> 0x234
```

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
```

SBREL

Syntax

```
SBREL expr
```

Precedence

1

Description

The expression must evaluate to a symbol `s`, with a possible addend `a`, written as `s+a`.

The result is the offset to `s` from the addressing origin of its output segment, plus the addend.

The result is `s+a-base`, if the origin of the output segment for `s` is `base`.

The normal usage is for the symbol `s` to be in `.data`, and to have base address of `.data` in `R9`.

Example

`SBREL(array+4)->4` if `array` is placed first in its output segment.

SFB section begin

Syntax

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

Precedence

1

Parameters

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

Description

`SFB` accepts a single operand to its right. The 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 `SIZEOF` is used.

Description

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

Example

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

`Table.s`:

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

`Application.s`:

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

UGT Unsigned greater than

Precedence

7

Description

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

Example

```
2 UGT 1 -> 1
-1 UGT 1 -> 1
```

ULT Unsigned less than

Precedence

7

Description

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

Example

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

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.

☞ The precedence of `:LEOR:` is 8.

Example

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

Assembler directives

Contents

Summary of assembler directives 73
Description of assembler directives 76
- Module control directives 76
- Symbol control directives 78
- Mode control directives 80
- Section control directives 81
- Value assignment directives 84
- Conditional assembly directives 86
- Macro processing directives 87
- Listing control directives 95
- C-style preprocessor directives 98
- Data definition or allocation directives 102
- Assembler control directives 105
- Function directives 108
- Call frame information directives for names blocks 108
- Call frame information directives for common blocks 109
- Call frame information directives for data blocks 110
- Call frame information directives for tracking resources and CFAs 111
- Call frame information directives for stack usage analysis 113

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 76*
- *Symbol control directives, page 78*
- *Mode control directives, page 80*
- *Section control directives, page 81*
- *Value assignment directives, page 84*
- *Conditional assembly directives, page 86*
- *Macro processing directives, page 87*
- *Listing control directives, page 95*
- *C-style preprocessor directives, page 98*
- *Data definition or allocation directives, page 102*
- *Assembler control directives, page 105*
- *Function directives, page 108*
- *Call frame information directives for names blocks, page 108*.
- *Call frame information directives for common blocks, page 109*
- *Call frame information directives for data blocks, page 110*
- *Call frame information directives for tracking resources and CFAs, page 111*
- *Call frame information directives for stack usage analysis, page 113*

This table gives a summary of all the assembler directives:

Directive	Description	Section
`_args`	Is set to number of arguments passed to macro.	Macro processing
`$`	Includes a file.	Assembler control
`#define`	Assigns a value to a label.	C-style preprocessor
`#elif`	Introduces a new condition in an `#if`...`#endif` block.	C-style preprocessor
`#else`	Assembles instructions if a condition is false.	C-style preprocessor
`#endif`	Ends an `#if`, `#ifdef`, or `#ifndef` block.	C-style preprocessor
`#error`	Generates an error.	C-style preprocessor
`#if`	Assembles instructions if a condition is true.	C-style preprocessor
`#ifdef`	Assembles instructions if a symbol is defined.	C-style preprocessor
`#ifndef`	Assembles instructions if a symbol is undefined.	C-style preprocessor
`#include`	Includes a file.	C-style preprocessor
`#message`	Generates a message on standard output.	C-style preprocessor
`#pragma`	Recognized but ignored.	C-style preprocessor
`#undef`	Undefines a label.	C-style preprocessor
`/*comment*/`	C-style comment delimiter.	Assembler control
`//`	C++ style comment delimiter.	Assembler control
`=`	Assigns a permanent value local to a module.	Value assignment
`AAPCS`	Sets module attributes.	Module control
`ALIAS`	Assigns a permanent value local to a module.	Value assignment
`ALIGN`	Aligns the program location counter by inserting zero-filled bytes.	Section control
`ALIGNRAM`	Aligns the program location counter.	Section control
`ALIGNROM`	Aligns the program location counter by inserting zero-filled bytes.	Section control
`ARM`	Interprets subsequent instructions as 32-bit (Arm) instructions.	Mode control
`ASEGN`	Begins a named absolute segment.	Segment control
`ASSIGN`	Assigns a temporary value.	Value assignment
`CALL_GRAPH_ROOT`	Specifies that a function is a call graph root.	Function
`CASEOFF`	Disables case sensitivity.	Assembler control
`CASEON`	Enables case sensitivity.	Assembler control
`CFI`	Specifies call frame information.	Call frame information
`CODE`	Interprets subsequent instructions as Arm, Thumb, or A64 instructions, depending on the setting of related assembler options.	Mode control
`CODE16`	Interprets subsequent instructions as 16-bit (Thumb) instructions. Replaced by `THUMB`.	Mode control
`CODE32`	Interprets subsequent instructions as 32-bit (Arm) instructions. Replaced by `ARM`.	Mode control
`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
`DATA64`	Defines an area with 64-bit alignment data within a code section.	Mode control

Directive	Description	Section
DC8	Generates 8-bit constants, including strings.	Data definition or allocation
DC16	Generates 16-bit constants.	Data definition or allocation
DC24	Generates 24-bit constants.	Data definition or allocation
DC32	Generates 32-bit constants.	Data definition or allocation
DC64	Generates 64-bit constants.	Data definition or allocation
DCB	Generates 8-bit byte constants, including strings.	Data definition or allocation
DCD	Generates 32-bit long word constants. Alias for DC32.	Data definition or allocation
DCQ	Generates 64-bit long constants. Alias for DC64.	Data definition or allocation
DCW	Generates 16-bit word constants, including strings. Alias for DC16.	Data definition or allocation
DEFINE	Defines a file-wide value.	Value assignment
DF16	Generates 16-bit half precision floating point constants.	Data definition or allocation
DF32	Generates 32-bit floating-point constants.	Data definition or allocation
DF64	Generates 64-bit floating-point constants.	Data definition or allocation
DS8	Allocates space for 8-bit integers.	Data definition or allocation
DS16	Allocates space for 16-bit integers.	Data definition or allocation
DS24	Allocates space for 24-bit integers.	Data definition or allocation
DS32	Allocates space for 32-bit integers.	Data definition or allocation
DS64	Allocates space for 64-bit integers.	Data definition or allocation
ELSE	Assembles instructions if a condition is false.	Conditional assembly
ELSEIF	Specifies a new condition in an IF...ENDIF block.	Conditional assembly
END	Ends the assembly of the last module in a file.	Module control
ENDIF	Ends an IF block.	Conditional assembly
ENDM	Ends a macro definition.	Macro processing
ENDR	Ends a repeat structure.	Macro processing
EQU	Assigns a permanent value local to a module.	Value assignment
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
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

Directive	Description	Section
`LTORG`	Directs the current literal pool to be assembled immediately following the directive.	Assembler control
`MACRO`	Defines a macro.	Macro processing
`ODD`	Aligns the program location counter to an odd address.	Section control
`OVERLAY`	Recognized but ignored.	Symbol control
`PAGE`	Retained for backward compatibility reasons.	Listing control
`PAGSIZ`	Retained for backward compatibility reasons.	Listing control
`PRESERVE8`	Sets a module attribute.	Module control
`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 15. Assembler directives summary

DESCRIPTION OF ASSEMBLER DIRECTIVES

The following pages give reference information about the assembler directives.

Module control directives

Syntax

```
AAPCS [modifier [...]]

END

PRESERVE8

REQUIRE8

RTMODEL key, value
```

Parameters

key	A text string specifying the key.
modifier	An AAPCS extension—possible values are `INTERWORK`, `VFP`, `VFP_COMPATIBLE`, `ROPI`, `RWPI`, `RWPI_COMPATIBLE`. Modifiers can be combined to specify AAPCS variants.
value	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 and types to them. For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions, page 23*.

Directive	Description	Expression restrictions
`END`	Ends the assembly of the module in a file.	Locally defined symbols plus offset or integer constants
`RTMODEL`	Declares runtime model attributes.	Not applicable

Table 16. Module control directives

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

In 32-bit mode, 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.

It is up to you to verify that the module is compliant to these parts, as the assembler does not verify this. In 64-bit mode, these directives have no effect.

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.

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 define 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

EXTERN *symbol* [,*symbol*] …

EXTWEAK *symbol* [,*symbol*] …

IMPORT *symbol* [,*symbol*] …

PUBLIC *symbol* [,*symbol*] …

PUBWEAK *symbol* [,*symbol*] …

```
REQUIRE symbol
```

Parameters

symbol	Symbol to be imported or exported.

Description

These directives control how symbols are shared between modules:

Directive	Description
`EXTERN`, `IMPORT`	Imports an external symbol.
`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 17. Symbol control directives

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 the linker. If a module containing a `PUBLIC` definition of a symbol is linked with one or more modules containing `PUBWEAK` definitions of the same symbol, the linker uses the `PUBLIC` definition.

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

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

Mode control directives

Syntax

```
ARM

CODE

CODE16

CODE32

DATA

DATA64

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 `--arm`, `--cpu_mode`, or `--thumb`, or as A64 instructions in AArch64 state, if any of the options `--aarch64`, `--abi`, or `--cpu_mode a64` have been used.
`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.
`DATA64`	Defines an area of 64-bit aligned data within a code section, where labels work as in a CODE area for the A64 instruction set in the AArch64 state.
`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 18. 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 that there is no way of changing between the Arm or Thumb processor modes to the A64 instruction set in the AArch64 state, or back.

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, page 126* 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*]

ASEGN *section* [: *type*] [: *flag*] [, *address*]

EVEN [*value*]

ODD [*value*]

RSEG *section* [: *type*] [: *flag*] [(*align*)]

SECTION *section* [: *type*] [: *flag*] [(*align*)]

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

Parameters

address	Address where this section part is placed.
align	The power of two to which the address should be aligned. The permitted range is 0 to 8. The default align value is 0, except for code sections where the default is 1.
flag	ROOT, NOROOT ROOT (the default mode) indicates that the section fragment must not be discarded. NOROOT means that the section fragment is discarded by the linker if no symbols in this section fragment are referred to. Normally, all section fragments except startup code and interrupt vectors should set this flag. REORDER, NOREORDER NOREORDER (the default mode) starts a new fragment in the section with the given name, or a new section if no such section exists. REORDER starts a new section with the given name.
section	The name of the section. The section name is a user-defined symbol that follows the rules described in *Symbols, page 18*.
type	The memory type, which can be either CODE, CONST, or DATA.
value	4-byte value used for padding. The default is zero. The padding demand determines how many bytes are used from the value, starting from the lowest byte.
type-expr	A constant expression that identifies the ELF type of the section.
flags-expr	A constant expression that identifies 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 23*.

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
ASEGN	Begins a named absolute section.	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

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 19. Section control directives

Beginning a named absolute section

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

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

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.

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, 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

const_expr	Constant value assigned to symbol.
expr	Value assigned to symbol or value to be tested.
label	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 20. 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

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

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 isand `ENDIF` directives are optional, and if used, 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 23*.

Directive	Description	Expression restrictions
`_args`	Is set to number of arguments passed to macro.	
`ENDM`	Ends a macro definition.	
`ENDR`	Ends a repeat structure.	
`EXITM`	Exits prematurely from a macro.	
`LOCAL`	Creates symbols local to a macro.	
`MACRO`	Defines a macro.	

Directive	Description	Expression restrictions
`REPT`	Assembles instructions a specified number of times.	No forward references No external references Absolute Fixed
`REPTC`	Repeats and substitutes characters.	
`REPTI`	Repeats and substitutes text.	

Table 21. 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.

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 47*.

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.

Double quotes have a special meaning—their only use is to enclose the characters to iterate over. Single quotes have no special meaning and are treated as ordinary characters.

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

```
COL columns

LSTCND{+|-}

LSTCOD{+|-}

LSTEXP{+|-}

LSTMAC{+|-}

LSTOUT{+|-}

LSTPAG{+|-}

LSTREP{+|-}

LSTXRF{+|-}

PAGE

PAGSIZ lines
```

Parameters

columns	An absolute expression in the range 80 to 132. The default is 80.
lines	An absolute expression in the range 10 to 150. The 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 22. 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

condition	An absolute assembler expression, see *Expressions, operands, and operators, page 16*. The expression must not contain any assembler labels or symbols, and any non-zero value is considered as true. The C preprocessor operator `defined` can be used.
filename	Name of file to be included or referred.
message	Text to be displayed.
symbol	Preprocessor symbol to be defined, undefined, or tested.
text	Value to be assigned.

Description

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

The preprocessor expressions use the same precedence rules as the assembler operators.

These C-language preprocessor directives are available:

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

Table 23. 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, as an assembler directive is not necessarily accepted as a part of the C preprocessor language.

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, 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 `R0` is decremented by an amount that depends on `adjust`, for example 30 when `adjust` is 3.

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

#define     tweak  1
#define     adjust 3

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

            end
```

Including 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 44*.

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.

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

DC8 *expr* [,*expr*] ...

DC16 *expr* [,*expr*] ...

DC24 *expr* [,*expr*] ...

DC32 *expr* [,*expr*] ...

DC64 *expr* [,*expr*] ...

DCB *expr* [,*expr*] ...

DCD *expr* [,*expr*] ...

DCQ *expr* [,*expr*] ...

DCW *expr* [,*expr*] ...

DF16 *value* [,*value*] ...

DF32 *value* [,*value*] ...

```
DF64 value [,value] ...

DS  count

DS8  count

DS16  count

DS24  count

DS32  count

DS64  count
```

Parameters

count	A valid absolute expression specifying the number of elements to be reserved.
expr	A valid absolute, relocatable, or external expression, or an ASCII string. ASCII strings are zero filled to a multiple of the data size implied by the directive. Double-quoted strings are zero-terminated. For `DC64`, *expr* cannot be relocatable or external.
value	A valid absolute expression or floating-point constant.

Description

These directives define values or reserve memory.

Use `DC8`, `DC16`, `DC24`, `DC32`, `DC64`,`DCB`, `DCD`, `DCQ`, `DCW`, `DF16`, `DF32`, or `DF64` to create a constant, which means an area of bytes is reserved big enough for the constant.

Use `DS8`, `DS16`, `DS24`, `DS32`, or `DS64` to reserve a number of uninitialized bytes.

For information about the restrictions that apply when using a directive in an expression, see *Expression restrictions, page 23*.

The column *Alias* in the following table shows the Arm Limited directive that corresponds to the IAR directive.

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.
`DC64`	`DCQ`	Generates 64-bit constants.
`DF16`		Generates 16-bit floating-point constants.
`DF32`		Generates 32-bit floating-point constants.
`DF64`		Generates 64-bit floating-point constants.
`DS8`	`DS`	Allocates space for 8-bit integers.
`DS16`		Allocates space for 16-bit integers.
`DS24`		Allocates space for 24-bit integers.
`DS32`		Allocates space for 32-bit integers.
`DS64`		Allocates space for 64-bit integers.

Table 24. Data definition or allocation directives

Relocatable expressions cannot be used in a DC8 directive.

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

```
$filename

/* comment */

// comment

CASEOFF

CASEON

INCLUDE filename

LTORG

RADIX expr
```

Parameters

comment	Comment ignored by the assembler.
expr	Default base—default 10 (decimal).
filename	Name of file to be included. The `$` character must be the first character on the line.

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 *Expression restrictions, page 23*.

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

Table 25. Assembler control directives

Use `$` to insert the contents of a file into the source file at a specified point. `$filename` is an alias for `#include "filename"`, see the section *Including source files* under *C-style preprocessor directives, page 98*. The `$` character must be the first character on the line.

Use `/*...*/` to comment sections of the assembler listing.

Use `//` to mark the rest of the line as comment.

Use INCLUDE to insert the contents of a file into the source file at a specified point. INCLUDE *filename* is an alias for #include <*filename*>, see the section *Including source files* under *C-style preprocessor directives, page 98*. Note that INCLUDE only searches in the system header directories.

Use LTORG to direct where the current literal pool is to be assembled. By default, this is performed at every END and RSEG directive. For an example, see *LDR (ARM), page 119*.

Use RADIX to set the default base for constants. The default base is 10.

Controlling case sensitivity

Use CASEON or CASEOFF to turn case sensitivity on or off 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 the section *Comments in C-style preprocessor directives* under *C-style preprocessor directives, page 98*.

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

```
CALL_GRAPH_ROOT function [,category]
```

Parameters

function	The function, a symbol.
category	An optional call graph root category, a string.

Description

Use this directive to specify that, for stack usage analysis purposes, the function `function` is a call graph root. You can also specify an optional category, a quoted string.

The compiler will generate this directive in assembler list files, when needed.

Example

```
CALL_GRAPH_ROOT my_interrupt, "interrupt"
```

See also

Call frame information directives for stack usage analysis, page 113, 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 resourcepart, part[, part] …

CFI STACKFRAME cfa resource type [, cfa resource type] …

CFI BASEADDRESS cfa type [, cfa type] …
```

Parameters

bits	The size of the resource in bits.
cfa	The name of a CFA (canonical frame address).
name	The name of the block.
namesblock	The name of a previously defined names block.
offset	The offset relative the CFA. An integer with an optional sign.
part	A part of a composite resource. The name of a previously declared resource.
resource	The name of a resource.

size	The size of the frame cell in bytes.
type	The segment memory type, such as `CODE`, `CONST`, or `DATA`. In addition, any of the memory types supported by the 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 26. Call frame information directives names block

Example

Examples of using CFI directives, page 31

See also

Tracking call frame usage, page 25

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.
dataalignfactor	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.

name	The name of the block.
namesblock	The name of a previously defined names block.
resource	The name of a resource.
type	The memory type, such as `CODE`, `CONST`, or `DATA`. In addition, any of the segment memory types supported by the 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 27. 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 111*.

Example

Examples of using CFI directives, page 31

See also

Tracking call frame usage, page 25

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 28. 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 111*.

Example

Examples of using CFI directives, page 31

See also

Tracking call frame usage, page 25

Call frame information directives for tracking resources and CFAs

Syntax

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

CFI cfacfiexpr

CFI resource { UNDEFINED | SAMEVALUE | CONCAT }

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

CFI resourcecfiexpr
```

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`	*cfiexpr*	Performs a bitwise `NOT` on a CFI expression.
`LITERAL`	*expr*	Get the value of the assembler expression. This can insert the value of a regular assembler expression into a CFI expression.
`NOT`	*cfiexpr*	Negates a logical CFI expression.
`UMINUS`	*cfiexpr*	Performs arithmetic negation on a CFI expression.

Table 29. Unary operators in CFI expressions

Binary operators

Overall syntax: `OPERATOR(operand1,operand2)`

CFI operator	Operands	Description
`ADD`	*cfiexpr,cfiexpr*	Addition
`AND`	*cfiexpr,cfiexpr*	Bitwise `AND`
`DIV`	*cfiexpr,cfiexpr*	Division
`EQ`	*cfiexpr,cfiexpr*	Equal to
`GE`	*cfiexpr,cfiexpr*	Greater than or equal to
`GT`	*cfiexpr,cfiexpr*	Greater than
`LE`	*cfiexpr,cfiexpr*	Less than or equal to
`LSHIFT`	*cfiexpr,cfiexpr*	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`	*cfiexpr,cfiexpr*	Less than
`MOD`	*cfiexpr,cfiexpr*	Modulo
`MUL`	*cfiexpr,cfiexpr*	Multiplication
`NE`	*cfiexpr,cfiexpr*	Not equal to
`OR`	*cfiexpr,cfiexpr*	Bitwise OR
`RSHIFTA`	*cfiexpr,cfiexpr*	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`	*cfiexpr,cfiexpr*	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`	*cfiexpr,cfiexpr*	Subtraction
`XOR`	*cfiexpr,cfiexpr*	Bitwise `XOR`

Table 30. Binary operators in CFI expressions

Ternary operators

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

Operator	Operands	Description
`FRAME`	*cfa,size,offset*	Gets the value from a stack frame. The operands are:

Operator	Operands	Description
		cfa, an identifier that denotes a previously declared CFA. *size*, a constant expression that denotes a size in bytes. *offset*, a constant expression that denotes a size in bytes. Gets the value at address *cfa+offset* of size *size*.
IF	*cond*,*true*,*false*	Conditional operator. The operands are: *cond*, a CFI expression that denotes a condition. *true*, any CFI expression. *false*, any CFI expression. If the conditional expression is non-zero, the result is the value of the *true* expression; otherwise the result is the value of the *false* expression.
LOAD	*size*,*type*,*addr*	Gets the value from memory. The operands are: *size*, a constant expression that denotes a size in bytes. *type*, a memory type. *addr*, a CFI expression that denotes a memory address. Gets the value at address *addr* in the segment memory type *type* of size *size*.

Table 31. Ternary operators in CFI expressions

Description

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

Directive	Description
CFI *cfa*	Declares the value of a CFA.
CFI *resource*	Declares the value of a resource.

Table 32. Call frame information directives for tracking resources and CFAs

Example

Examples of using CFI directives, page 31

See also

Tracking call frame usage, page 25

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 33. Call frame information directives for stack usage analysis

See also

Tracking call frame usage, page 25

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

Assembler pseudo-instructions

Contents

Summary 115
Descriptions of pseudo-instructions 116
ADR (ARM) 116
ADR (CODE16) 117
ADR (THUMB) 117
ADRL (64-bit mode) 118
ADRL (ARM) 118
ADRL (THUMB) 119
LDR (64-bit mode) 119
LDR (ARM) 119
LDR (CODE16) 120
LDR (THUMB) 120
MOV (CODE16) 121
MOV32 (THUMB) 122
MOVL (64-bit mode) 122
NOP (ARM) 123
NOP (CODE16) 123

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.

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

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

This is a summary of the available pseudo-instructions for the A32 and T32 instruction sets:

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.

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 34. Pseudo-instructions for A32 and T32

* Deprecated. Use THUMB instead.

This is a summary of the available pseudo-instructions for the A64 instruction set:

Pseudo-instruction	Translated to	Description
ADRL	ADRP, ADD	Loads a program-relative address into a register.
LDR	LDR	Loads a register with any 32-bit expression.
MOVL	MOVZ(, MOVK)	Loads a register with a 32-bit or 64-bit value.

Table 35. Pseudo-instructions for A64

DESCRIPTIONS OF PSEUDO-INSTRUCTIONS

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

ADR (ARM)

Syntax

```
ADR{condition} register,expression
```

Parameters

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

Description

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

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

            thumb
thumbLabel  ; ...
```

```
                end
```

ADR (CODE16)

Syntax

```
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
            alignrom 2
dataLabel   dc32    0xABCD19

            end
```

See also

ADR (CODE16), page 117 if only 16-bit Thumb instructions are available.

ADRL (64-bit mode)

Syntax

```
ADRL register,expression
```

Parameters

register	The register to load, `X0`–`X30`.
expression	A program-relative expression that evaluates to an address within the range -4 Gbytes to +4 Gbytes from the instruction.

Description

The `ADRL` pseudo-instruction loads a program-relative address into a register. It is similar to the `ADR` instruction. `ADRL` can load a wider range of addresses than `ADR` because it generates two data processing instructions. The assembler will not attempt to determine if the expression is within range. If the address is not within the range -4 Gbytes to +4 Gbytes, the linker will generate an error message and linking will fail.

ADRL (ARM)

Syntax

```
ADRL{condition} register,expression
```

Parameters

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

Description

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

Example

```
            name    armAdrL
            section MYCODE:CODE(2)

            arm
            adrl    r1,label+0x2345 ; Becomes "add r1,pc,#0x45"
                                    ;     and "add r1,r1,#0x2300"
            data
label       dc32    0

            end
```

ADRL (THUMB)

Syntax

```
ADRL{condition} register,expression
```

Parameters

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

Description

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

LDR (64-bit mode)

Syntax

```
LDRregister,=expression
```

Parameters

register	The register to load, `X0`–`X30` or `W0`–`W30`.
expression	Any 32-bit or 64-bit expression. A 64-bit expression can be loaded into `X0`–`X30` and a 32-bit expression can be loaded into `W0`–`W30`.

Description

The `LDR` pseudo-instruction loads a register with any 32-bit or 64-bit expression. 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 instruction to the constant must be within the range -1 Mbyte to +1 Mbyte.

LDR (ARM)

Syntax

```
LDR{condition} register,=expression1
```

or

```
LDR{condition} register,expression2
```

Parameters

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

Description

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

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

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
                                    ; here.
            end
```

See also

`LTORG` in *Assembler control directives, page 105*.

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 `MOVS` 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

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

Parameters

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

Description

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

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

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

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

As for `LDR (CODE16)`, the 16-bit variant can be translated into a `MOVS` instruction, which modifies the program status register.

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

Example

```
            name    thumbLdr
            extern  extLabel

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

See also

LDR (CODE16), page 120 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.

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.

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

Example

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

MOV32 (THUMB)

Syntax

```
MOV32{condition} register,expression
```

Parameters

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

Description

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

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

MOVL (64-bit mode)

Syntax

```
MOVL register,#expression
```

Parameters

register	The register to load, `X0–X30` or `W0–W30`.
expression	Any 32-bit or 64-bit expression. A 64-bit expression can be loaded into `X0–X30` and a 32-bit expression can be loaded into `W0–W30`.

Description

Similar to the `LDR` pseudo-instruction, but will load the constant by generating a pair of a `MOV` instruction and a `MOVK` instruction (for a 32-bit expression), or a `MOV` instruction and three `MOVK` instructions (for a 64-bit expression). Unresolved expressions are not supported.

NOP (ARM)

Syntax

```
NOP
```

Description

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

```
MOV r0,r0
```

`NOP` is not a pseudo-instruction in architecture versions that include a `NOP` instruction (Armv6K, Armv6T2, Armv7, Armv8-M).

NOP (CODE16)

Syntax

```
NOP
```

Description

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

```
MOV r8,r8
```

`NOP` is not a pseudo-instruction in architecture versions that include a `NOP` instruction (Armv6T2, Armv7, Armv8-M).

Assembler diagnostics

Contents

Message format 124
Severity levels 124
Options for diagnostics 124
Assembler warning messages 124
Command line error messages 124
Assembler error messages 125
Assembler fatal error messages 125
Assembler internal error messages 125

The following pages describe the format of 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 54*
- Set the number of maximum errors before the assembly stops, see *-E, page 41*.

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 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

Contents

Introduction 126
Thumb code labels 126
Alternative register names 127
Alternative mnemonics 127
Operator synonyms 128
Warning messages 129
The first register operand omitted 129
The first register operand duplicated 129
Immediate #0 omitted in Load/Store 129

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 using an existing product to using the IAR Assembler for Arm in 32-bit mode.

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.

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

The instructions and descriptions in this chapter apply only to using the IAR Assembler for Arm in 32-bit mode.

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 alternative register names and 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
SL	R10
FP	R11
IP	R12

Table 36. Alternative register names

For further descriptions of the registers, see *Register symbols, page 18*.

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

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

Table 37. Alternative mnemonics

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:`	`&`

Operator synonym	Assembler operator
`:EOR:`	`^`
`:LAND:`	`&&`
`:LEOR:`	`XOR`
`:LNOT:`	`!`
`:LOR:`	`||`
`:MOD:`	`%`
`:NOT:`	`~`
`:OR:`	`|`
`:SHL:`	`<<`
`:SHR:`	`>>`

Table 38. Operator synonyms

In some cases, assembler operators and operator synonyms have different precedence levels. For further descriptions of the operators, see *Assembler operators, page 57*.

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]
```