# Arm® Compiler

**Version 6.11**

# armasm User Guide

# Arm® Compiler

## armasm User Guide

### Release Information

**Document History**

| Issue | Date | Confidentiality | Change |
|---|---|---|---|
| A | 14 March 2014 | Non-Confidential | Arm Compiler v6.00 Release |
| B | 15 December 2014 | Non-Confidential | Arm Compiler v6.01 Release |
| C | 30 June 2015 | Non-Confidential | Arm Compiler v6.02 Release |
| D | 18 November 2015 | Non-Confidential | Arm Compiler v6.3 Release |
| E | 24 February 2016 | Non-Confidential | Arm Compiler v6.4 Release |
| F | 29 June 2016 | Non-Confidential | Arm Compiler v6.5 Release |
| G | 04 November 2016 | Non-Confidential | Arm Compiler v6.6 Release |
| 0607-00 | 05 April 2017 | Non-Confidential | Arm Compiler v6.7 Release. Document numbering scheme has changed. |
| 0608-00 | 30 July 2017 | Non-Confidential | Arm Compiler v6.8 Release. |
| 0609-00 | 25 October 2017 | Non-Confidential | Arm Compiler v6.9 Release. |
| 0610-00 | 14 March 2018 | Non-Confidential | Arm Compiler v6.10 Release. |
| 0610-01 | 01 June 2018 | Non-Confidential | Document update to include the CSDB instruction for the Arm Compiler v6.10 Release. |
| 0611-00 | 25 October 2018 | Non-Confidential | Arm Compiler v6.11 Release. Removed the A32/T32 and A64 instruction set descriptions. |

**Confidentiality Status**

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

**Product Status**

The information in this document is Final, that is for a developed product.

**Web Address**

*http://www.arm.com*

Contents
# Arm® Compiler armasm User Guide

# List of Figures
# **Arm® Compiler armasm User Guide**

# List of Tables
# Arm® Compiler armasm User Guide

# Preface

This preface introduces the *Arm® Compiler armasm User Guide*.

It contains the following:

## About this book

Arm® Compiler armasm User Guide. This document provides information for using the Arm legacy assembler (armasm). It contains information on command-line options, assembler directives, and supports the Armv7 and Armv8 architectures.

### Using this book

This book is organized into the following chapters:

***Chapter 1 Overview of the Assembler***
Gives an overview of the assemblers provided with Arm Compiler toolchain.

***Chapter 2 Structure of Assembly Language Modules***
Describes the structure of assembly language source files.

***Chapter 3 Writing A32/T32 Assembly Language***
Describes the use of a few basic A32 and T32 instructions and the use of macros.

***Chapter 4 Using armasm***
Describes how to use `armasm`.

***Chapter 5 armasm Command-line Options***
Describes the `armasm` command-line syntax and command-line options.

***Chapter 6 Symbols, Literals, Expressions, and Operators***
Describes how you can use symbols to represent variables, addresses, and constants in code, and how you can combine these with operators to create numeric or string expressions.

***Chapter 7 Directives Reference***
Describes the directives that are provided by the Arm assembler, `armasm`.

***Chapter 8 armasm-Specific Instruction Set Features***
Describes the additional support that `armasm` provides for the Arm instruction set.

***Chapter 9 Via File Syntax***
Describes the syntax of via files accepted by `armasm`.

### Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

### Typographic conventions

*italic*
Introduces special terminology, denotes cross-references, and citations.

**bold**
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

<u>`mono`</u>`space`
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

*`monospace italic`*
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

**`monospace bold`**
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

## Feedback

### Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:
- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

### Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm Compiler armasm User Guide*.
- The number 100069_0611_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

**Note**

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

## Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

# Chapter 1
# **Overview of the Assembler**

Gives an overview of the assemblers provided with Arm Compiler toolchain.

It contains the following sections:

## 1.1 About the Arm® Compiler toolchain assemblers

The Arm Compiler toolchain provides different assemblers.

They are:

- The freestanding legacy assembler, `armasm`. Use `armasm` to assemble existing A64, A32, and T32 assembly language code written in armasm syntax.
- The `armclang` integrated assembler. Use this to assemble assembly language code written in GNU syntax.
- An optimizing inline assembler built into `armclang`. Use this to assemble assembly language code written in GNU syntax that is used inline in C or C++ source code.

**Note**

This book only applies to `armasm`. For information on `armclang`, see the *armclang Reference Guide*.

**Note**

Be aware of the following:

- Generated code might be different between two Arm Compiler releases.
- For a feature release, there might be significant code generation differences.

**Note**

The command-line option descriptions and related information in the individual Arm Compiler tools documents describe all the features that Arm Compiler supports. Any features not documented are not supported and are used at your own risk. You are responsible for making sure that any generated code using *community features* on page 1-23 is operating correctly.

***Related information***

*Arm Compiler armclang Reference Guide*

*Mixing Assembly Code with C or C++ Code*

*Assembling armasm and GNU syntax assembly code*

## 1.2 Key features of the armasm assembler

The `armasm` assembler supports instructions, directives, and user-defined macros.

It supports:

- *Unified Assembly Language* (UAL) for both A32 and T32 code.
- Assembly language for A64 code.
- Advanced SIMD instructions in A64, A32, and T32 code.
- Floating-point instructions in A64, A32, and T32 code.
- Directives in assembly source code.
- Processing of user-defined macros.
- `SDOT` and `UDOT` instructions are an optional extension in Armv8.2-A and later, and a mandtory extension in Armv8.4-A and later.

**Note**

`armasm` does not support some architectural features, such as:

- Half-precision floating-point multiply with add or multiply with subtract arithmetic operations. These instructions are an optional extension in Armv8.2-A and Armv8.3-A, and a mandatory extension in Armv8.4-A and later. See `+fp16fml` in the *-mcpu* command-line option in the *armclang Reference Guide*.
- AArch64 Crypto instructions (for SHA512, SHA3, SM3, SM4). See `+crypto` in the *-mcpu* command-line option in the *armclang Reference Guide*.

***Related concepts***

*1.3 How the assembler works* on page 1-19

*3.1 About the Unified Assembler Language* on page 3-36

*3.22 Use of macros* on page 3-63

***Related reference***

*Chapter 7 Directives Reference* on page 7-197

*5.13 --cpu=name* on page 5-109

***Related information***

*-mcpu*

*Arm Compiler Instruction Set Reference Guide*

## 1.3 How the assembler works

`armasm` reads the assembly language source code twice before it outputs object code. Each read of the source code is called a pass.

This is because assembly language source code often contains forward references. A forward reference occurs when a label is used as an operand, for example as a branch target, earlier in the code than the definition of the label. The assembler cannot know the address of the forward reference label until it reads the definition of the label.

During each pass, the assembler performs different functions. In the first pass, the assembler:

- Checks the syntax of the instruction or directive. It faults if there is an error in the syntax, for example if a label is specified on a directive that does not accept one.
- Determines the size of the instruction and data being assembled and reserves space.
- Determines offsets of labels within sections.
- Creates a symbol table containing label definitions and their memory addresses.

In the second pass, the assembler:

- Faults if an undefined reference is specified in an instruction operand or directive.
- Encodes the instructions using the label offsets from pass 1, where applicable.
- Generates relocations.
- Generates debug information if requested.
- Outputs the object file.

Memory addresses of labels are determined and finalized in the first pass. Therefore, the assembly code must not change during the second pass. All instructions must be seen in both passes. Therefore you must not define a symbol after a `:DEF:` test for the symbol. The assembler faults if it sees code in pass 2 that was not seen in pass 1.

### Line not seen in pass 1

The following example shows that `num EQU 42` is not seen in pass 1 but is seen in pass 2:

```
    AREA x,CODE
    [ :DEF: foo
num EQU 42
    ]
foo DCD num
    END
```

Assembling this code generates the error:

```
A1903E: Line not seen in first pass; cannot be assembled.
```

### Line not seen in pass 2

The following example shows that `MOV r1,r2` is seen in pass 1 but not in pass 2:

```
    AREA x,CODE
    [ :LNOT: :DEF: foo
    MOV r1, r2
    ]
foo MOV r3, r4
    END
```

Assembling this code generates the error:

```
A1909E: Line not seen in second pass; cannot be assembled.
```

***Related concepts***

*4.13 Two pass assembler diagnostics* on page 4-88

*3.25 Instruction and directive relocations* on page 3-67

***Related reference***

*1.4 Directives that can be omitted in pass 2 of the assembler* on page 1-21

*5.17 --diag_error=tag[,tag,...]* on page 5-115
*5.14 --debug* on page 5-112

## 1.4 Directives that can be omitted in pass 2 of the assembler

Most directives must appear in both passes of the assembly process. You can omit some directives from the second pass over the source code by the assembler, but doing this is strongly discouraged.

Directives that can be omitted from pass 2 are:

- `GBLA`, `GBLL`, `GBLS`.
- `LCLA`, `LCLL`, `LCLS`.
- `SETA`, `SETL`, `SETS`.
- `RN`, `RLIST`.
- `CN`, `CP`.
- `SN`, `DN`, `QN`.
- `EQU`.
- `MAP`, `FIELD`.
- `GET`, `INCLUDE`.
- `IF`, `ELSE`, `ELIF`, `ENDIF`.
- `WHILE`, `WEND`.
- `ASSERT`.
- `ATTR`.
- `COMMON`.
- `EXPORTAS`.
- `IMPORT`.
- `EXTERN`.
- `KEEP`.
- `MACRO`, `MEND`, `MEXIT`.
- `REQUIRE8`.
- `PRESERVE8`.

**Note**

Macros that appear only in pass 1 and not in pass 2 must contain only these directives.

### ASSERT directive appears in pass 1 only

The code in the following example assembles without error although the `ASSERT` directive does not appear in pass 2:

```
    AREA ||.text||,CODE
x   EQU 42
    IF :LNOT: :DEF: sym
        ASSERT x == 42
    ENDIF
sym EQU 1
    END
```

### Use of ELSE and ELIF directives

Directives that appear in pass 2 but do not appear in pass 1 cause an assembly error. However, this does not cause an assembly error when using the `ELSE` and `ELIF` directives if their matching `IF` directive appears in pass 1. The following example assembles without error because the `IF` directive appears in pass 1:

```
    AREA ||.text||,CODE
x   EQU 42
    IF :DEF: sym
    ELSE
        ASSERT x == 42
    ENDIF
sym EQU 1
    END
```

***Related concepts***

*1.3 How the assembler works* on page 1-19

*4.13 Two pass assembler diagnostics* on page 4-88

## 1.5 Support level definitions

This describes the levels of support for various Arm Compiler 6 features.

Arm Compiler 6 is built on Clang and LLVM technology. Therefore it has more functionality than the set of product features described in the documentation. The following definitions clarify the levels of support and guarantees on functionality that are expected from these features.

Arm welcomes feedback regarding the use of all Arm Compiler 6 features, and endeavors to support users to a level that is appropriate for that feature. You can contact support at *https://developer.arm.com/support*.

### Identification in the documentation

All features that are documented in the Arm Compiler 6 documentation are product features, except where explicitly stated. The limitations of non-product features are explicitly stated.

### Product features

Product features are suitable for use in a production environment. The functionality is well-tested, and is expected to be stable across feature and update releases.

- Arm endeavors to give advance notice of significant functionality changes to product features.
- If you have a support and maintenance contract, Arm provides full support for use of all product features.
- Arm welcomes feedback on product features.
- Any issues with product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

In addition to fully supported product features, some product features are only alpha or beta quality.

**Beta product features**

Beta product features are implementation complete, but have not been sufficiently tested to be regarded as suitable for use in production environments.

Beta product features are indicated with .

- Arm endeavors to document known limitations on beta product features.
- Beta product features are expected to eventually become product features in a future release of Arm Compiler 6.
- Arm encourages the use of beta product features, and welcomes feedback on them.
- Any issues with beta product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

**Alpha product features**

Alpha product features are not implementation complete, and are subject to change in future releases, therefore the stability level is lower than in beta product features.

Alpha product features are indicated with .

- Arm endeavors to document known limitations of alpha product features.
- Arm encourages the use of alpha product features, and welcomes feedback on them.
- Any issues with alpha product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

### Community features

Arm Compiler 6 is built on LLVM technology and preserves the functionality of that technology where possible. This means that there are additional features available in Arm Compiler that are not listed in the documentation. These additional features are known as community features. For information on these community features, see the *documentation for the Clang/LLVM project*.

Where community features are referenced in the documentation, they are indicated with .

- Arm makes no claims about the quality level or the degree of functionality of these features, except when explicitly stated in this documentation.
- Functionality might change significantly between feature releases.
- Arm makes no guarantees that community features will remain functional across update releases, although changes are expected to be unlikely.

Some community features might become product features in the future, but Arm provides no roadmap for this. Arm is interested in understanding your use of these features, and welcomes feedback on them. Arm supports customers using these features on a best-effort basis, unless the features are unsupported. Arm accepts defect reports on these features, but does not guarantee that these issues will be fixed in future releases.

### Guidance on use of community features

There are several factors to consider when assessing the likelihood of a community feature being functional:

- The following figure shows the structure of the Arm Compiler 6 toolchain:

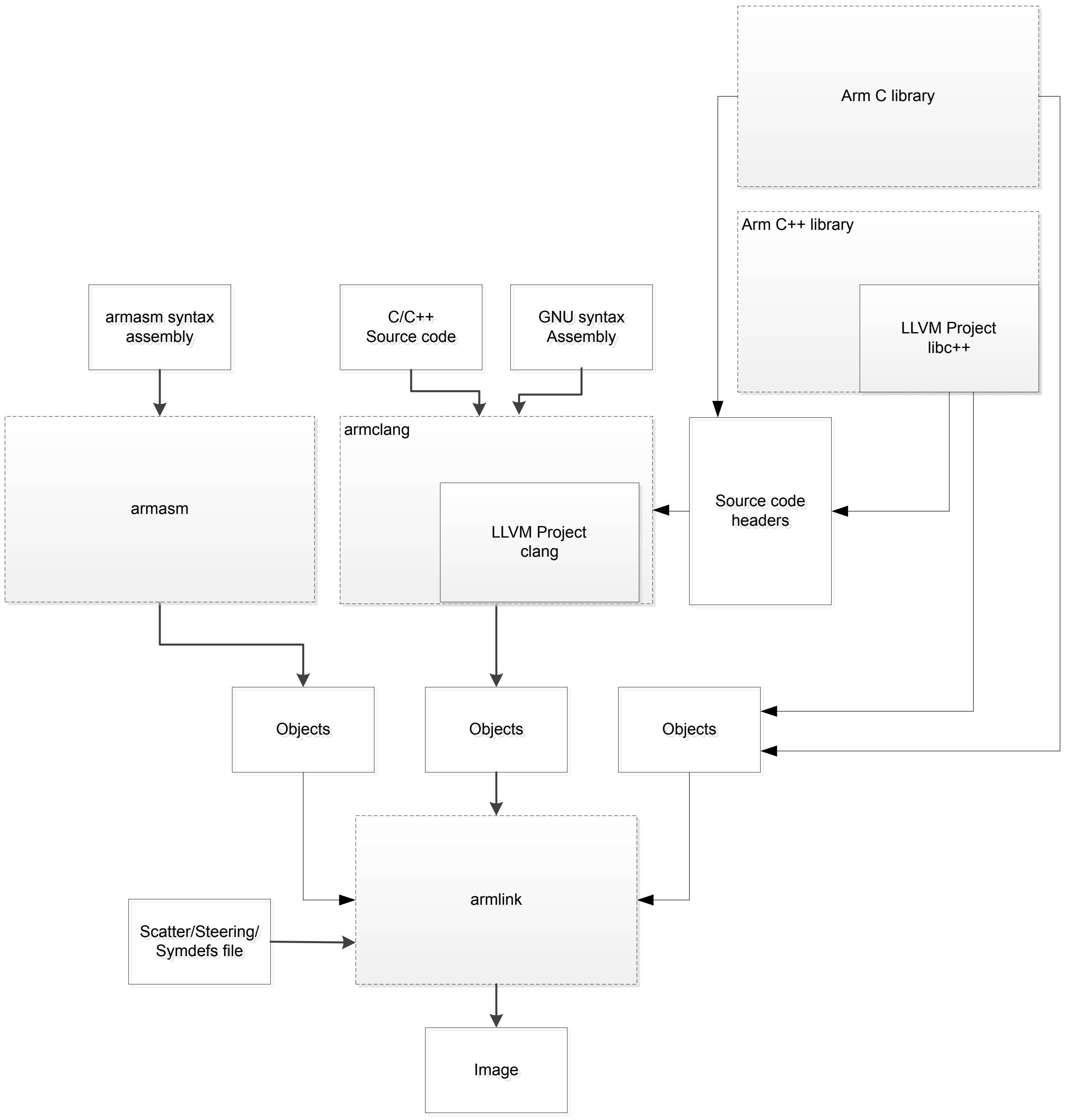

**Figure 1-1 Integration boundaries in Arm Compiler 6.**

The dashed boxes are toolchain components, and any interaction between these components is an integration boundary. Community features that span an integration boundary might have significant limitations in functionality. The exception to this is if the interaction is codified in one of the standards supported by Arm Compiler 6. See *Application Binary Interface (ABI) for the Arm® Architecture*. Community features that do not span integration boundaries are more likely to work as expected.

- Features primarily used when targeting hosted environments such as Linux or BSD might have significant limitations, or might not be applicable, when targeting bare-metal environments.
- The Clang implementations of compiler features, particularly those that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support

for new language features, is likely to be less mature and therefore more likely to have limited functionality.

## Unsupported features

With both the product and community feature categories, specific features and use-cases are known not to function correctly, or are not intended for use with Arm Compiler 6.

Limitations of product features are stated in the documentation. Arm cannot provide an exhaustive list of unsupported features or use-cases for community features. The known limitations on community features are listed in *Community features* on page 1-23.

### List of known unsupported features

The following is an incomplete list of unsupported features, and might change over time:

- The Clang option `-stdlib=libstdc++` is not supported.
- C++ static initialization of local variables is not thread-safe when linked against the standard C++ libraries. For thread-safety, you must provide your own implementation of thread-safe functions as described in *Standard C++ library implementation definition*.

  **Note**

  This restriction does not apply to the [ALPHA]-supported multithreaded C++ libraries.

- Use of C11 library features is unsupported.
- Any community feature that exclusively pertains to non-Arm architectures is not supported.
- Compilation for targets that implement architectures older than Armv7 or Armv6-M is not supported.
- The **`long double`** data type is not supported for AArch64 state because of limitations in the current Arm C library.
- Complex numbers are not supported because of limitations in the current Arm C library.

# Chapter 2
# Structure of Assembly Language Modules

Describes the structure of assembly language source files.

It contains the following sections:

## 2.1 Syntax of source lines in assembly language

The assembler parses and assembles assembly language to produce object code.

### Syntax

Each line of assembly language source code has this general form:

`{symbol} {instruction|directive|pseudo-instruction} {;comment}`

All three sections of the source line are optional.

*symbol* is usually a label. In instructions and pseudo-instructions it is always a label. In some directives it is a symbol for a variable or a constant. The description of the directive makes this clear in each case.

*symbol* must begin in the first column. It cannot contain any white space character such as a space or a tab unless it is enclosed by bars (|).

Labels are symbolic representations of addresses. You can use labels to mark specific addresses that you want to refer to from other parts of the code. Numeric local labels are a subclass of labels that begin with a number in the range 0-99. Unlike other labels, a numeric local label can be defined many times. This makes them useful when generating labels with a macro.

Directives provide important information to the assembler that either affects the assembly process or affects the final output image.

Instructions and pseudo-instructions make up the code a processor uses to perform tasks.

**Note**

Instructions, pseudo-instructions, and directives must be preceded by white space, such as a space or a tab, irrespective of whether there is a preceding label or not.

Some directives do not allow the use of a label.

---

A comment is the final part of a source line. The first semicolon on a line marks the beginning of a comment except where the semicolon appears inside a string literal. The end of the line is the end of the comment. A comment alone is a valid line. The assembler ignores all comments. You can use blank lines to make your code more readable.

### Considerations when writing assembly language source code

You must write instruction mnemonics, pseudo-instructions, directives, and symbolic register names (except `a1-a4` and `v1-v8` in A32 or T32 instructions) in either all uppercase or all lowercase. You must not use mixed case. Labels and comments can be in uppercase, lowercase, or mixed case.

```
        AREA     A32ex, CODE, READONLY
                                ; Name this block of code A32ex

        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0x123456      ; AArch32 semihosting (formerly SWI)
        END                     ; Mark end of file
```

To make source files easier to read, you can split a long line of source into several lines by placing a backslash character (\) at the end of the line. The backslash must not be followed by any other characters, including spaces and tabs. The assembler treats the backslash followed by end-of-line sequence as white space. You can also use blank lines to make your code more readable.

———— **Note** ————

Do not use the backslash followed by end-of-line sequence within quoted strings.

————————————

The limit on the length of lines, including any extensions using backslashes, is 4095 characters.

***Related concepts***

*6.6 Labels* on page 6-173

*6.10 Numeric local labels* on page 6-177

*6.13 String literals* on page 6-180

***Related reference***

*2.2 Literals* on page 2-30

*6.1 Symbol naming rules* on page 6-168

*6.15 Syntax of numeric literals* on page 6-182

## 2.2 Literals

Assembly language source code can contain numeric, string, Boolean, and single character literals.

Literals can be expressed as:

- Decimal numbers, for example `123`.
- Hexadecimal numbers, for example `0x7B`.
- Numbers in any base from `2` to `9`, for example `5_204` is a number in base `5`.
- Floating point numbers, for example `123.4`.
- Boolean values `{TRUE}` or `{FALSE}`.
- Single character values enclosed by single quotes, for example `'w'`.
- Strings enclosed in double quotes, for example `"This is a string"`.

——— **Note** ———

In most cases, a string containing a single character is accepted as a single character value. For example `ADD r0,r1,#"a"` is accepted, but `ADD r0,r1,#"ab"` is faulted.

———

You can also use variables and names to represent literals.

***Related reference***

*2.1 Syntax of source lines in assembly language* on page 2-28

## 2.3 ELF sections and the AREA directive

Object files produced by the assembler are divided into sections. In assembly source code, you use the `AREA` directive to mark the start of a section.

ELF sections are independent, named, indivisible sequences of code or data. A single code section is the minimum required to produce an application.

The output of an assembly or compilation can include:

- One or more code sections. These are usually read-only sections.
- One or more data sections. These are usually read-write sections. They might be *zero-initialized* (ZI).

The linker places each section in a program image according to section placement rules. Sections that are adjacent in source files are not necessarily adjacent in the application image

Use the `AREA` directive to name the section and set its attributes. The attributes are placed after the name, separated by commas.

You can choose any name for your sections. However, names starting with any non-alphabetic character must be enclosed in bars, or an `AREA name missing` error is generated. For example, `|1_DataArea|`.

The following example defines a single read-only section called `A32ex` that contains code:

```
        AREA  A32ex, CODE, READONLY ; Name this block of code A32ex
```

***Related concepts***

*2.4 An example armasm syntax assembly language module* on page 2-32

***Related reference***

*7.6 AREA* on page 7-205

***Related information***

*Information about scatter files*

## 2.4 An example armasm syntax assembly language module

An armasm syntax assembly language module has several constituent parts.

These are:
- ELF sections (defined by the `AREA` directive).
- Application entry (defined by the `ENTRY` directive).
- Application execution.
- Application termination.
- Program end (defined by the `END` directive).

### Constituents of an A32 assembly language module

The following example defines a single section called `A32ex` that contains code and is marked as being `READONLY`. This example uses the A32 instruction set.

```
        AREA     A32ex, CODE, READONLY
                                ; Name this block of code A32ex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0x123456      ; AArch32 semihosting (formerly SWI)
        END                     ; Mark end of file
```

### Constituents of an A64 assembly language module

The following example defines a single section called `A64ex` that contains code and is marked as being `READONLY`. This example uses the A64 instruction set.

```
        AREA     A64ex, CODE, READONLY
                                ; Name this block of code A64ex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      w0, #10        ; Set up parameters
        MOV      w1, #3
        ADD      w0, w0, w1     ; w0 = w0 + w1
stop
        MOV      x1, #0x26
        MOVK     x1, #2, LSL #16
        STR      x1, [sp,#0]    ; ADP_Stopped_ApplicationExit
        MOV      x0, #0
        STR      x0, [sp,#8]    ; Exit status code
        MOV      x1, sp         ; x1 contains the address of parameter block
        MOV      w0, #0x18      ; angel_SWIreason_ReportException
        HLT      0xf000         ; AArch64 semihosting
        END                     ; Mark end of file
```

### Constituents of a T32 assembly language module

The following example defines a single section called `T32ex` that contains code and is marked as being `READONLY`. This example uses the T32 instruction set.

```
        AREA     T32ex, CODE, READONLY
                                ; Name this block of code T32ex
        ENTRY                   ; Mark first instruction to execute
        THUMB
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0xab          ; AArch32 semihosting (formerly SWI)
        ALIGN    4              ; Aligned on 4-byte boundary
        END                     ; Mark end of file
```

### Application entry

The `ENTRY` directive declares an entry point to the program. It marks the first instruction to be executed. In applications using the C library, an entry point is also contained within the C library initialization code. Initialization code and exception handlers also contain entry points.

### Application execution in A32 or T32 code

The application code begins executing at the label `start`, where it loads the decimal values 10 and 3 into registers `R0` and `R1`. These registers are added together and the result placed in `R0`.

### Application execution in A64 code

The application code begins executing at the label `start`, where it loads the decimal values 10 and 3 into registers `W0` and `W1`. These registers are added together and the result placed in `W0`.

### Application termination

After executing the main code, the application terminates by returning control to the debugger.

**A32 and T32 code**

You do this in A32 and T32 code using the semihosting `SVC` instruction:

- In A32 code, the semihosting `SVC` instruction is `0x123456` by default.
- In T32 code, use the semihosting `SVC` instruction is `0xAB` by default.

A32 and T32 code uses the following parameters:

- R0 equal to `angel_SWIreason_ReportException` (`0x18`).
- R1 equal to `ADP_Stopped_ApplicationExit` (`0x20026`).

**A64 code**

In A64 code, use `HLT` instruction `0xF000` to invoke the semihosting interface.

A64 code uses the following parameters:

- W0 equal to `angel_SWIreason_ReportException` (`0x18`).
- X1 is the address of a block of two parameters. The first is the exception type, `ADP_Stopped_ApplicationExit` (`0x20026`) and the second is the exit status code.

### Program end

The `END` directive instructs the assembler to stop processing this source file. Every assembly language source module must finish with an `END` directive on a line by itself. Any lines following the `END` directive are ignored by the assembler.

***Related concepts***

*2.3 ELF sections and the AREA directive* on page 2-31

***Related reference***

*7.23 END* on page 7-225

*7.25 ENTRY* on page 7-227

***Related information***

*Semihosting for AArch32 and AArch64*

# Chapter 3
# Writing A32/T32 Assembly Language

Describes the use of a few basic A32 and T32 instructions and the use of macros.

It contains the following sections:

## 3.1 About the Unified Assembler Language

*Unified Assembler Language* (UAL) is a common syntax for A32 and T32 instructions. It supersedes earlier versions of both the A32 and T32 assembler languages.

Code that is written using UAL can be assembled for A32 or T32 for any Arm processor. `armasm` faults the use of unavailable instructions.

`armasm` can assemble code that is written in pre-UAL and UAL syntax.

By default, `armasm` expects source code to be written in UAL. `armasm` accepts UAL syntax if any of the directives `CODE32`, `ARM`, or `THUMB` is used or if you assemble with any of the `--32`, `--arm`, or `--thumb` command-line options. `armasm` also accepts source code that is written in pre-UAL A32 assembly language when you assemble with the `CODE32` or `ARM` directive.

`armasm` accepts source code that is written in pre-UAL T32 assembly language when you assemble using the `--16` command-line option, or the `CODE16` directive in the source code.

——— **Note** ———

The pre-UAL T32 assembly language does not support 32-bit T32 instructions.

———————————

***Related reference***

*5.1 --16* on page 5-96
*7.7 ARM or CODE32 directive* on page 7-209
*7.11 CODE16 directive* on page 7-213
*7.65 THUMB directive* on page 7-275
*5.2 --32* on page 5-97
*5.4 --arm* on page 5-100
*5.59 --thumb* on page 5-157

## 3.2 Syntax differences between UAL and A64 assembly language

UAL is the assembler syntax that is used by the A32 and T32 instruction sets. A64 assembly language is the assembler syntax that is used by the A64 instruction set.

UAL in Armv8 is unchanged from Armv7.

The general statement format and operand order of A64 assembly language is the same as UAL, but there are some differences between them. The following table describes the main differences:

**Table 3-1 Syntax differences between UAL and A64 assembly language**

| UAL | A64 |
|---|---|
| You make an instruction conditional by appending a condition code suffix directly to the mnemonic, with no delimiter. For example: `BEQ label` | For conditionally executed instructions, you separate the condition code suffix from the mnemonic using a `.` delimiter. For example: `B.EQ label` |
| Apart from the `IT` instruction, there are no unconditionally executed integer instructions that use a condition code as an operand. | A64 provides several unconditionally executed instructions that use a condition code as an operand. For these instructions, you specify the condition code to test for in the final operand position. For example: `CSEL w1,w2,w3,EQ` |
| The `.W` and `.N` instruction width specifiers control whether the assembler generates a 32-bit or 16-bit encoding for a T32 instruction. | A64 is a fixed width 32-bit instruction set so does not support `.W` and `.N` qualifiers. |
| The core register names are R0-R15. | Qualify register names to indicate the operand data size, either 32-bit (W0-W31) or 64-bit (X0-X31). |
| You can refer to registers R13, R14, and R15 as synonyms for SP, LR, and PC respectively. | In AArch64, there is no register that is named W31 or X31. Instead, you can refer to register 31 as SP, WZR, or XZR, depending on the context. You cannot refer to PC either by name or number. LR is an alias for register 30. |
| A32 has no equivalent of the extend operators. | You can specify an extend operator in several instructions to control how a portion of the second source register value is sign or zero extended. For example, in the following instruction, `UXTB` is the extend type (zero extend, byte) and `#2` is an optional left shift amount: `ADD X1, X2, W3, UXTB #2` |

## 3.3 Register usage in subroutine calls

You use branch instructions to call and return from subroutines. The Procedure Call Standard for the Arm Architecture defines how to use registers in subroutine calls.

A subroutine is a block of code that performs a task based on some arguments and optionally returns a result. By convention, you use registers R0 to R3 to pass arguments to subroutines, and R0 to pass a result back to the callers. A subroutine that requires more than four inputs uses the stack for the additional inputs.

To call subroutines, use a branch and link instruction. The syntax is:

```
    BL  destination
```

where `destination` is usually the label on the first instruction of the subroutine.

`destination` can also be a PC-relative expression.

The `BL` instruction:

- Places the return address in the link register.
- Sets the PC to the address of the subroutine.

After the subroutine code has executed you can use a `BX LR` instruction to return.

**Note**

Calls between separately assembled or compiled modules must comply with the restrictions and conventions defined by the *Procedure Call Standard for the Arm® Architecture*.

### Example

The following example shows a subroutine, `doadd`, that adds the values of two arguments and returns a result in `R0`:

```
        AREA    subrout, CODE, READONLY     ; Name this block of code
        ENTRY                      ; Mark first instruction to execute
start   MOV     r0, #10            ; Set up parameters
        MOV     r1, #3
        BL      doadd              ; Call subroutine
stop    MOV     r0, #0x18          ; angel_SWIreason_ReportException
        LDR     r1, =0x20026       ; ADP_Stopped_ApplicationExit
        SVC     #0x123456          ; AArch32 semihosting (formerly SWI)
doadd   ADD     r0, r0, r1         ; Subroutine code
        BX      lr                 ; Return from subroutine
        END                        ; Mark end of file
```

***Related concepts***

*3.17 Stack operations for nested subroutines* on page 3-57

***Related information***

*Procedure Call Standard for the Arm Architecture*

*Procedure Call Standard for the Arm 64-bit Architecture (AArch64)*

## 3.4 Load immediate values

To represent some immediate values, you might have to use a sequence of instructions rather than a single instruction.

A32 and T32 instructions can only be 32 bits wide. You can use a `MOV` or `MVN` instruction to load a register with an immediate value from a range that depends on the instruction set. Certain 32-bit values cannot be represented as an immediate operand to a single 32-bit instruction, although you can load these values from memory in a single instruction.

You can load any 32-bit immediate value into a register with two instructions, a `MOV` followed by a `MOVT`. Or, you can use a pseudo-instruction, `MOV32`, to construct the instruction sequence for you.

You can also use the `LDR` pseudo-instruction to load immediate values into a register.

You can include many commonly-used immediate values directly as operands within data processing instructions, without a separate load operation. The range of immediate values that you can include as operands in 16-bit T32 instructions is much smaller.

***Related concepts***

*3.5 Load immediate values using MOV and MVN* on page 3-40
*3.6 Load immediate values using MOV32* on page 3-43
*3.7 Load immediate values using LDR Rd, =const* on page 3-44

***Related reference***

*8.5 LDR pseudo-instruction* on page 8-285

## 3.5 Load immediate values using MOV and MVN

The `MOV` and `MVN` instructions can write a range of immediate values to a register.

In A32:

- `MOV` can load any 8-bit immediate value, giving a range of `0x0`-`0xFF` (0-255).

  It can also rotate these values by any even number.

  These values are also available as immediate operands in many data processing operations, without being loaded in a separate instruction.
- `MVN` can load the bitwise complements of these values. The numerical values are `-(`*n*`+1)`, where *n* is the value available in `MOV`.
- `MOV` can load any 16-bit number, giving a range of `0x0`-`0xFFFF` (0-65535).

The following table shows the range of 8-bit values that can be loaded in a single A32 `MOV` or `MVN` instruction (for data processing operations). The value to load must be a multiple of the value shown in the Step column.

**Table 3-2 A32 state immediate values (8-bit)**

| Binary | Decimal | Step | Hexadecimal | MVN value[a] | Notes |
|---|---|---|---|---|---|
| `000000000000000000000000abcdefgh` | 0-255 | 1 | 0-`0xFF` | -1 to -256 | - |
| `0000000000000000000000abcdefgh00` | 0-1020 | 4 | 0-`0x3FC` | -4 to -1024 | - |
| `00000000000000000000abcdefgh0000` | 0-4080 | 16 | 0-`0xFF0` | -16 to -4096 | - |
| `000000000000000000abcdefgh000000` | 0-16320 | 64 | 0-`0x3FC0` | -64 to -16384 | - |
| ... | ... | ... | ... | ... | - |
| `abcdefgh000000000000000000000000` | 0-255 x $2^{24}$ | $2^{24}$ | 0-`0xFF000000` | 1-256 x $-2^{24}$ | - |
| `cdefgh000000000000000000000000ab` | (bit pattern) | - | - | (bit pattern) | See b in Note |
| `efgh000000000000000000000000abcd` | (bit pattern) | - | - | (bit pattern) | See b in Note |
| `gh000000000000000000000000abcdef` | (bit pattern) | - | - | (bit pattern) | See b in Note |

The following table shows the range of 16-bit values that can be loaded in a single `MOV` A32 instruction:

**Table 3-3 A32 state immediate values in MOV instructions**

| Binary | Decimal | Step | Hexadecimal | MVN value | Notes |
|---|---|---|---|---|---|
| `0000000000000000abcdefghijklmnop` | 0-65535 | 1 | 0-`0xFFFF` | - | See c in Note |

**Note**

These notes give extra information on both tables.

**a**
The `MVN` values are only available directly as operands in `MVN` instructions.

**b**
These values are available in A32 only. All the other values in this table are also available in 32-bit T32 instructions.

**c**
These values are not available directly as operands in other instructions.

In T32:

- The 32-bit `MOV` instruction can load:
  - Any 8-bit immediate value, giving a range of `0x0-0xFF` (0-255).
  - Any 8-bit immediate value, shifted left by any number.
  - Any 8-bit pattern duplicated in all four bytes of a register.
  - Any 8-bit pattern duplicated in bytes 0 and 2, with bytes 1 and 3 set to 0.
  - Any 8-bit pattern duplicated in bytes 1 and 3, with bytes 0 and 2 set to 0.

  These values are also available as immediate operands in many data processing operations, without being loaded in a separate instruction.
- The 32-bit `MVN` instruction can load the bitwise complements of these values. The numerical values are `-(n+1)`, where *n* is the value available in `MOV`.
- The 32-bit `MOV` instruction can load any 16-bit number, giving a range of `0x0-0xFFFF` (0-65535). These values are not available as immediate operands in data processing operations.

In architectures with T32, the 16-bit T32 `MOV` instruction can load any immediate value in the range 0-255.

The following table shows the range of values that can be loaded in a single 32-bit T32 `MOV` or `MVN` instruction (for data processing operations). The value to load must be a multiple of the value shown in the Step column.

**Table 3-4 32-bit T32 immediate values**

| Binary | Decimal | Step | Hexadecimal | MVN value[a] | Notes |
|---|---|---|---|---|---|
| `000000000000000000000000abcdefgh` | 0-255 | 1 | `0x0-0xFF` | -1 to -256 | - |
| `00000000000000000000000abcdefgh0` | 0-510 | 2 | `0x0-0x1FE` | -2 to -512 | - |
| `0000000000000000000000abcdefgh00` | 0-1020 | 4 | `0x0-0x3FC` | -4 to -1024 | - |
| ... | ... | ... | ... | ... | - |
| `0abcdefgh00000000000000000000000` | 0-255 x $2^{23}$ | $2^{23}$ | `0x0-0x7F800000` | 1-256 x -$2^{23}$ | - |
| `abcdefgh000000000000000000000000` | 0-255 x $2^{24}$ | $2^{24}$ | `0x0-0xFF000000` | 1-256 x -$2^{24}$ | - |
| `abcdefghabcdefghabcdefghabcdefgh` | (bit pattern) | - | `0xXYXYXYXY` | `0xXYXYXYXY` | - |
| `00000000abcdefgh00000000abcdefgh` | (bit pattern) | - | `0x00XY00XY` | `0xFFXYFFXY` | - |
| `abcdefgh00000000abcdefgh00000000` | (bit pattern) | - | `0xXY00XY00` | `0xXYFFXYFF` | - |
| `00000000000000000000abcdefghijkl` | 0-4095 | 1 | `0x0-0xFFF` | - | See b in Note |

The following table shows the range of 16-bit values that can be loaded by the `MOV` 32-bit T32 instruction:

**Table 3-5 32-bit T32 immediate values in MOV instructions**

| Binary | Decimal | Step | Hexadecimal | MVN value | Notes |
|---|---|---|---|---|---|
| `0000000000000000abcdefghijklmnop` | 0-65535 | 1 | `0x0-0xFFFF` | - | See c in Note |

**Note**

These notes give extra information on the tables.

**a**

The `MVN` values are only available directly as operands in `MVN` instructions.

**b**

These values are available directly as operands in `ADD`, `SUB`, and `MOV` instructions, but not in `MVN` or any other data processing instructions.

**c**

These values are only available in `MOV` instructions.

---

In both A32 and T32, you do not have to decide whether to use `MOV` or `MVN`. The assembler uses whichever is appropriate. This is useful if the value is an assembly-time variable.

If you write an instruction with an immediate value that is not available, the assembler reports the error: `Immediate` *`n`* `out of range for this operation.`

***Related concepts***

*3.4 Load immediate values* on page 3-39

## 3.6 Load immediate values using MOV32

To load any 32-bit immediate value, a pair of `MOV` and `MOVT` instructions is equivalent to a `MOV32` pseudo-instruction.

Both A32 and T32 instruction sets include:

- A `MOV` instruction that can load any value in the range `0x00000000` to `0x0000FFFF` into a register.
- A `MOVT` instruction that can load any value in the range `0x0000` to `0xFFFF` into the most significant half of a register, without altering the contents of the least significant half.

You can use these two instructions to construct any 32-bit immediate value in a register. Alternatively, you can use the `MOV32` pseudo-instruction. The assembler generates the `MOV`, `MOVT` instruction pair for you.

You can also use the `MOV32` instruction to load addresses into registers by using a label or any PC-relative expression in place of an immediate value. The assembler puts a relocation directive into the object file for the linker to resolve the address at link-time.

***Related concepts***

*6.5 Register-relative and PC-relative expressions* on page 6-172

***Related reference***

*8.6 MOV32 pseudo-instruction* on page 8-287

## 3.7 Load immediate values using LDR Rd, =const

The `LDR Rd,=const` pseudo-instruction generates the most efficient single instruction to load any 32-bit number.

You can use this pseudo-instruction to generate constants that are out of range of the `MOV` and `MVN` instructions.

The `LDR` pseudo-instruction generates the most efficient single instruction for the specified immediate value:

- If the immediate value can be constructed with a single `MOV` or `MVN` instruction, the assembler generates the appropriate instruction.
- If the immediate value cannot be constructed with a single `MOV` or `MVN` instruction, the assembler:
  - Places the value in a *literal pool* (a portion of memory embedded in the code to hold constant values).
  - Generates an `LDR` instruction with a PC-relative address that reads the constant from the literal pool.

  For example:

  ```
      LDR      rn, [pc, #offset to literal pool]
                             ; load register n with one word
                             ; from the address [pc + offset]
  ```

  You must ensure that there is a literal pool within range of the `LDR` instruction generated by the assembler.

***Related concepts***

*3.8 Literal pools* on page 3-45

***Related reference***

*8.5 LDR pseudo-instruction* on page 8-285

## 3.8 Literal pools

The assembler uses literal pools to store some constant data in code sections. You can use the LTORG directive to ensure a literal pool is within range.

The assembler places a literal pool at the end of each section. The end of a section is defined either by the END directive at the end of the assembly or by the AREA directive at the start of the following section. The END directive at the end of an included file does not signal the end of a section.

In large sections the default literal pool can be out of range of one or more LDR instructions. The offset from the PC to the constant must be:

- Less than 4KB in A32 or T32 code when the 32-bit LDR instruction is available, but can be in either direction.
- Forward and less than 1KB when only the 16-bit T32 LDR instruction is available.

When an LDR Rd,=const pseudo-instruction requires the immediate value to be placed in a literal pool, the assembler:

- Checks if the value is available and addressable in any previous literal pools. If so, it addresses the existing constant.
- Attempts to place the value in the next literal pool if it is not already available.

If the next literal pool is out of range, the assembler generates an error message. In this case you must use the LTORG directive to place an additional literal pool in the code. Place the LTORG directive after the failed LDR pseudo-instruction, and within the valid range for an LDR instruction.

You must place literal pools where the processor does not attempt to execute them as instructions. Place them after unconditional branch instructions, or after the return instruction at the end of a subroutine.

### Example of placing literal pools

The following example shows the placement of literal pools. The instructions listed as comments are the A32 instructions generated by the assembler.

```
        AREA     Loadcon, CODE, READONLY
        ENTRY                      ; Mark first instruction to execute
start
        BL       func1             ; Branch to first subroutine
        BL       func2             ; Branch to second subroutine
stop
        MOV      r0, #0x18         ; angel_SWIreason_ReportException
        LDR      r1, =0x20026      ; ADP_Stopped_ApplicationExit
        SVC      #0x123456         ; AArch32 semihosting (formerly SWI)func1
        LDR      r0, =42           ; => MOV R0, #42
        LDR      r1, =0x55555555   ; => LDR R1, [PC, #offset to
                                   ; Literal Pool 1]
        LDR      r2, =0xFFFFFFFF   ; => MVN R2, #0
        BX       lr
        LTORG                      ; Literal Pool 1 contains
                                   ; literal Ox55555555
func2
        LDR      r3, =0x55555555   ; => LDR R3, [PC, #offset to
                                   ; Literal Pool 1]
        ; LDR r4, =0x66666666      ; If this is uncommented it
                                   ; fails, because Literal Pool 2
                                   ; is out of reach
        BX       lr
LargeTable
        SPACE    4200              ; Starting at the current location,
                                   ; clears a 4200 byte area of memory
                                   ; to zero
        END                        ; Literal Pool 2 is inserted here,
                                   ; but is out of range of the LDR
                                   ; pseudo-instruction that needs it
```

***Related concepts***

*3.7 Load immediate values using LDR Rd, =const* on page 3-44

***Related reference***

*7.50 LTORG* on page 7-255

## 3.9 Load addresses into registers

It is often necessary to load an address into a register. There are several ways to do this.

For example, you might have to load the address of a variable, a string literal, or the start location of a jump table.

Addresses are normally expressed as offsets from a label, or from the current PC or other register.

You can load an address into a register either:

- Using the instruction `ADR`.
- Using the pseudo-instruction `ADRL`.
- Using the pseudo-instruction `MOV32`.
- From a literal pool using the pseudo-instruction `LDR Rd,=Label`.

***Related concepts***

*3.10 Load addresses to a register using ADR* on page 3-47

*3.11 Load addresses to a register using ADRL* on page 3-49

*3.6 Load immediate values using MOV32* on page 3-43

*3.12 Load addresses to a register using LDR Rd, =label* on page 3-50

## 3.10 Load addresses to a register using ADR

The `ADR` instruction loads an address within a certain range, without performing a data load.

`ADR` accepts a PC-relative expression, that is, a label with an optional offset where the address of the label is relative to the PC.

**Note**

The label used with `ADR` must be within the same code section. The assembler faults references to labels that are out of range in the same section.

The available range of addresses for the `ADR` instruction depends on the instruction set and encoding:

**A32**

Any value that can be produced by rotating an 8-bit value right by any even number of bits within a 32-bit word. The range is relative to the PC.

**32-bit T32 encoding**

±4095 bytes to a byte, halfword, or word-aligned address.

**16-bit T32 encoding**

0 to 1020 bytes. *`label`* must be word-aligned. You can use the `ALIGN` directive to ensure this.

### Example of a jump table implementation with ADR

This example shows A32 code that implements a jump table. Here, the `ADR` instruction loads the address of the jump table.

```
        AREA   Jump, CODE, READONLY ; Name this block of code
        ARM                         ; Following code is A32 code
num     EQU    2                    ; Number of entries in jump table
        ENTRY                       ; Mark first instruction to execute
start                               ; First instruction to call
        MOV    r0, #0               ; Set up the three arguments
        MOV    r1, #3
        MOV    r2, #2
        BL     arithfunc            ; Call the function
stop
        MOV    r0, #0x18            ; angel_SWIreason_ReportException
        LDR    r1, =0x20026         ; ADP_Stopped_ApplicationExit
        SVC    #0x123456            ; AArch32 semihosting (formerly
SWI)arithfunc                         ; Label the function
        CMP    r0, #num             ; Treat function code as unsigned
                                    ; integer
        BXHS   lr                   ; If code is >= num then return
        ADR    r3, JumpTable        ; Load address of jump table
        LDR    pc, [r3,r0,LSL#2]    ; Jump to the appropriate routine
JumpTable
        DCD    DoAdd
        DCD    DoSub
DoAdd
        ADD    r0, r1, r2           ; Operation 0
        BX     lr                   ; Return
DoSub
        SUB    r0, r1, r2           ; Operation 1
        BX     lr                   ; Return
        END                         ; Mark the end of this file
```

In this example, the function `arithfunc` takes three arguments and returns a result in `R0`. The first argument determines the operation to be carried out on the second and third arguments:

**argument1=0**

Result = argument2 + argument3.

**argument1=1**

Result = argument2 – argument3.

The jump table is implemented with the following instructions and assembler directives:

**`EQU`**

Is an assembler directive. You use it to give a value to a symbol. In this example, it assigns the value 2 to *`num`*. When *`num`* is used elsewhere in the code, the value 2 is substituted. Using `EQU` in this way is similar to using `#define` to define a constant in C.

**`DCD`**

Declares one or more words of store. In this example, each `DCD` stores the address of a routine that handles a particular clause of the jump table.

**`LDR`**

The `LDR PC,[R3,R0,LSL#2]` instruction loads the address of the required clause of the jump table into the PC. It:

- Multiplies the clause number in `R0` by 4 to give a word offset.
- Adds the result to the address of the jump table.
- Loads the contents of the combined address into the PC.

***Related concepts***

*3.12 Load addresses to a register using LDR Rd, =label* on page 3-50

*3.11 Load addresses to a register using ADRL* on page 3-49

## 3.11 Load addresses to a register using ADRL

The `ADRL` pseudo-instruction loads an address within a certain range, without performing a data load. The range is wider than that of the `ADR` instruction.

`ADRL` accepts a PC-relative expression, that is, a label with an optional offset where the address of the label is relative to the current PC.

——— **Note** ———

The label used with `ADRL` must be within the same code section. The assembler faults references to labels that are out of range in the same section.

———————————

The assembler converts an `ADRL` *`rn,label`* pseudo-instruction by generating:
- Two data processing instructions that load the address, if it is in range.
- An error message if the address cannot be constructed in two instructions.

The available range depends on the instruction set and encoding.

**A32**

Any value that can be generated by two `ADD` or two `SUB` instructions. That is, any value that can be produced by the addition of two values, each of which is 8 bits rotated right by any even number of bits within a 32-bit word. The range is relative to the PC.

**32-bit T32 encoding**

±1MB to a byte, halfword, or word-aligned address.

**16-bit T32 encoding**

`ADRL` is not available.

***Related concepts***

*3.10 Load addresses to a register using ADR* on page 3-47

*3.12 Load addresses to a register using LDR Rd, =label* on page 3-50

## 3.12 Load addresses to a register using LDR Rd, =label

The `LDR Rd,=label` pseudo-instruction places an address in a literal pool and then loads the address into a register.

`LDR Rd,=label` can load any 32-bit numeric value into a register. It also accepts PC-relative expressions such as labels, and labels with offsets.

The assembler converts an `LDR Rd,=label` pseudo-instruction by:

- Placing the address of *label* in a literal pool (a portion of memory embedded in the code to hold constant values).
- Generating a PC-relative `LDR` instruction that reads the address from the literal pool, for example:

```
    LDR rn [pc, #offset_to_literal_pool]
                        ; load register n with one word
                        ; from the address [pc + offset]
```

  You must ensure that the literal pool is within range of the `LDR` pseudo-instruction that needs to access it.

### Example of loading using LDR Rd, =label

The following example shows a section with two literal pools. The final `LDR` pseudo-instruction needs to access the second literal pool, but it is out of range. Uncommenting this line causes the assembler to generate an error.

The instructions listed in the comments are the A32 instructions generated by the assembler.

```
        AREA   LDRlabel, CODE, READONLY
        ENTRY                     ; Mark first instruction to execute
start
        BL     func1              ; Branch to first subroutine
        BL     func2              ; Branch to second subroutine
stop
        MOV    r0, #0x18          ; angel_SWIreason_ReportException
        LDR    r1, =0x20026       ; ADP_Stopped_ApplicationExit
        SVC    #0x123456          ; AArch32 semihosting (formerly SWI)
func1
        LDR    r0, =start         ; => LDR r0,[PC, #offset into Literal Pool 1]
        LDR    r1, =Darea + 12    ; => LDR r1,[PC, #offset into Literal Pool 1]
        LDR    r2, =Darea + 6000  ; => LDR r2,[PC, #offset into Literal Pool 1]
        BX     lr                 ; Return
        LTORG                     ; Literal Pool 1
func2
        LDR    r3, =Darea + 6000  ; => LDR r3,[PC, #offset into Literal Pool 1]
                                  ; (sharing with previous literal)
        ; LDR   r4, =Darea + 6004 ; If uncommented, produces an error because
                                  ; Literal Pool 2 is out of range.
        BX     lr                 ; Return
Darea   SPACE  8000               ; Starting at the current location, clears
                                  ; a 8000 byte area of memory to zero.
        END                       ; Literal Pool 2 is automatically inserted
                                  ; after the END directive.
                                  ; It is out of range of all the LDR
                                  ; pseudo-instructions in this example.
```

### Example of string copy

The following example shows an A32 code routine that overwrites one string with another. It uses the `LDR` pseudo-instruction to load the addresses of the two strings from a data section. The following are particularly significant:

**`DCB`**

The `DCB` directive defines one or more bytes of store. In addition to integer values, `DCB` accepts quoted strings. Each character of the string is placed in a consecutive byte.

**LDR, STR**

The `LDR` and `STR` instructions use post-indexed addressing to update their address registers. For example, the instruction:

```
LDRB    r2,[r1],#1
```

loads `R2` with the contents of the address pointed to by `R1` and then increments `R1` by 1.

The example also shows how, unlike the `ADR` and `ADRL` pseudo-instructions, you can use the `LDR` pseudo-instruction with labels that are outside the current section. The assembler places a relocation directive in the object code when the source file is assembled. The relocation directive instructs the linker to resolve the address at link time. The address remains valid wherever the linker places the section containing the `LDR` and the literal pool.

```
        AREA    StrCopy, CODE, READONLY
        ENTRY                       ; Mark first instruction to execute
start
        LDR     r1, =srcstr         ; Pointer to first string
        LDR     r0, =dststr         ; Pointer to second string
        BL      strcopy             ; Call subroutine to do copy
stop
        MOV     r0, #0x18           ; angel_SWIreason_ReportException
        LDR     r1, =0x20026        ; ADP_Stopped_ApplicationExit
        SVC     #0x123456           ; AArch32 semihosting (formerly SWI)
strcopy
        LDRB    r2, [r1],#1         ; Load byte and update address
        STRB    r2, [r0],#1         ; Store byte and update address
        CMP     r2, #0              ; Check for zero terminator
        BNE     strcopy             ; Keep going if not
        MOV     pc,lr               ; Return
        AREA    Strings, DATA, READWRITE
srcstr  DCB     "First string - source",0
dststr  DCB     "Second string - destination",0
        END
```

***Related concepts***

*3.11 Load addresses to a register using ADRL* on page 3-49

*3.7 Load immediate values using LDR Rd, =const* on page 3-44

***Related reference***

*8.5 LDR pseudo-instruction* on page 8-285

*7.15 DCB* on page 7-217

## 3.13 Other ways to load and store registers

You can load and store registers using `LDR`, `STR` and `MOV` (register) instructions.

You can load any 32-bit value from memory into a register with an `LDR` data load instruction. To store registers into memory you can use the `STR` data store instruction.

You can use the `MOV` instruction to move any 32-bit data from one register to another.

***Related concepts***

*3.14 Load and store multiple register instructions* on page 3-53

*3.15 Load and store multiple register instructions in A32 and T32* on page 3-54

## 3.14 Load and store multiple register instructions

The A32 and T32 instruction sets include instructions that load and store multiple registers. These instructions can provide a more efficient way of transferring the contents of several registers to and from memory than using single register loads and stores.

Multiple register transfer instructions are most often used for block copy and for stack operations at subroutine entry and exit. The advantages of using a multiple register transfer instruction instead of a series of single data transfer instructions include:

- Smaller code size.
- A single instruction fetch overhead, rather than many instruction fetches.
- On uncached Arm processors, the first word of data transferred by a load or store multiple is always a nonsequential memory cycle, but all subsequent words transferred can be sequential memory cycles. Sequential memory cycles are faster in most systems.

——— **Note** ———

The lowest numbered register is transferred to or from the lowest memory address accessed, and the highest numbered register to or from the highest address accessed. The order of the registers in the register list in the instructions makes no difference.

You can use the `--diag_warning 1206` assembler command line option to check that registers in register lists are specified in increasing order.

———————————

***Related concepts***

*3.15 Load and store multiple register instructions in A32 and T32* on page 3-54
*3.16 Stack implementation using LDM and STM* on page 3-55
*3.17 Stack operations for nested subroutines* on page 3-57
*3.18 Block copy with LDM and STM* on page 3-58

## 3.15 Load and store multiple register instructions in A32 and T32

Instructions are available in both the A32 and T32 instruction sets to load and store multiple registers.

They are:

**`LDM`**

Load Multiple registers.

**`STM`**

Store Multiple registers.

**`PUSH`**

Store multiple registers onto the stack and update the stack pointer.

**`POP`**

Load multiple registers off the stack, and update the stack pointer.

In `LDM` and `STM` instructions:

- The list of registers loaded or stored can include:
  - In A32 instructions, any or all of R0-R12, SP, LR, and PC.
  - In 32-bit T32 instructions, any or all of R0-R12, and optionally LR or PC (`LDM` only) with some restrictions.
  - In 16-bit T32 instructions, any or all of R0-R7.
- The address must be word-aligned. It can be:
  - Incremented after each transfer.
  - Incremented before each transfer (A32 instructions only).
  - Decremented after each transfer (A32 instructions only).
  - Decremented before each transfer (not in 16-bit encoded T32 instructions).
- The base register can be either:
  - Updated to point to the next block of data in memory.
  - Left as it was before the instruction.

When the base register is updated to point to the next block in memory, this is called writeback, that is, the adjusted address is written back to the base register.

In `PUSH` and `POP` instructions:

- The stack pointer (SP) is the base register, and is always updated.
- The address is incremented after each transfer in `POP` instructions, and decremented before each transfer in `PUSH` instructions.
- The list of registers loaded or stored can include:
  - In A32 instructions, any or all of R0-R12, SP, LR, and PC.
  - In 32-bit T32 instructions, any or all of R0-R12, and optionally LR or PC (`POP` only) with some restrictions.
  - In 16-bit T32 instructions, any or all of R0-R7, and optionally LR (`PUSH` only) or PC (`POP` only).

——— **Note** ———

Use of SP in the list of registers in these A32 instructions is deprecated.

A32 `STM` and `PUSH` instructions that use PC in the list of registers, and A32 `LDM` and `POP` instructions that use both PC and LR in the list of registers are deprecated.

———————————

***Related concepts***

*3.14 Load and store multiple register instructions* on page 3-53

## 3.16 Stack implementation using LDM and STM

You can use the `LDM` and `STM` instructions to implement pop and push operations respectively. You use a suffix to indicate the stack type.

The load and store multiple instructions can update the base register. For stack operations, the base register is usually the stack pointer, SP. This means that you can use these instructions to implement push and pop operations for any number of registers in a single instruction.

The load and store multiple instructions can be used with several types of stack:

**Descending or ascending**

The stack grows downwards, starting with a high address and progressing to a lower one (a *descending* stack), or upwards, starting from a low address and progressing to a higher address (an *ascending* stack).

**Full or empty**

The stack pointer can either point to the last item in the stack (a *full* stack), or the next free space on the stack (an *empty* stack).

To make it easier for the programmer, stack-oriented suffixes can be used instead of the increment or decrement, and before or after suffixes. The following table shows the stack-oriented suffixes and their equivalent addressing mode suffixes for load and store instructions:

**Table 3-6 Stack-oriented suffixes and equivalent addressing mode suffixes**

| Stack-oriented suffix | For store or push instructions | For load or pop instructions |
|---|---|---|
| FD (Full Descending stack) | DB (Decrement Before) | IA (Increment After) |
| FA (Full Ascending stack) | IB (Increment Before) | DA (Decrement After) |
| ED (Empty Descending stack) | DA (Decrement After) | IB (Increment Before) |
| EA (Empty Ascending stack) | IA (Increment After) | DB (Decrement Before) |

The following table shows the load and store multiple instructions with the stack-oriented suffixes for the various stack types:

**Table 3-7 Suffixes for load and store multiple instructions**

| Stack type | Store | Load |
|---|---|---|
| Full descending | `STMFD` (`STMDB`, Decrement Before) | `LDMFD` (`LDM`, increment after) |
| Full ascending | `STMFA` (`STMIB`, Increment Before) | `LDMFA` (`LDMDA`, Decrement After) |
| Empty descending | `STMED` (`STMDA`, Decrement After) | `LDMED` (`LDMIB`, Increment Before) |
| Empty ascending | `STMEA` (`STM`, increment after) | `LDMEA` (`LDMDB`, Decrement Before) |

For example:

```
    STMFD    sp!, {r0-r5}  ; Push onto a Full Descending Stack
    LDMFD    sp!, {r0-r5}  ; Pop from a Full Descending Stack
```

**——— Note ———**

The *Procedure Call Standard for the Arm® Architecture* (AAPCS), and `armclang` always use a full descending stack.

The `PUSH` and `POP` instructions assume a full descending stack. They are the preferred synonyms for `STMDB` and `LDM` with writeback.

---

***Related concepts***

*3.14 Load and store multiple register instructions* on page 3-53

***Related information***

*Procedure Call Standard for the Arm Architecture*

## 3.17 Stack operations for nested subroutines

Stack operations can be very useful at subroutine entry and exit to avoid losing register contents if other subroutines are called.

At the start of a subroutine, any working registers required can be stored on the stack, and at exit they can be popped off again.

In addition, if the link register is pushed onto the stack at entry, additional subroutine calls can be made safely without causing the return address to be lost. If you do this, you can also return from a subroutine by popping the PC off the stack at exit, instead of popping the LR and then moving that value into the PC. For example:

```
subroutine  PUSH    {r5-r7,lr} ; Push work registers and lr
            ; code
            BL      somewhere_else
            ; code
            POP     {r5-r7,pc} ; Pop work registers and pc
```

***Related concepts***

*3.3 Register usage in subroutine calls* on page 3-38

*3.14 Load and store multiple register instructions* on page 3-53

***Related information***

*Procedure Call Standard for the Arm Architecture*

*Procedure Call Standard for the Arm 64-bit Architecture (AArch64)*

## 3.18 Block copy with LDM and STM

You can sometimes make code more efficient by using `LDM` and `STM` instead of `LDR` and `STR` instructions.

### Example of block copy without LDM and STM

The following example is an A32 code routine that copies a set of words from a source location to a destination a single word at a time:

```
        AREA  Word, CODE, READONLY  ; name the block of code
num     EQU   20                    ; set number of words to be copied
        ENTRY                       ; mark the first instruction called
start
        LDR   r0, =src              ; r0 = pointer to source block
        LDR   r1, =dst              ; r1 = pointer to destination block
        MOV   r2, #num              ; r2 = number of words to copy
wordcopy
        LDR   r3, [r0], #4          ; load a word from the source and
        STR   r3, [r1], #4          ; store it to the destination
        SUBS  r2, r2, #1            ; decrement the counter
        BNE   wordcopy              ; ... copy more
stop
        MOV   r0, #0x18             ; angel_SWIreason_ReportException
        LDR   r1, =0x20026          ; ADP_Stopped_ApplicationExit
        SVC   #0x123456             ; AArch32 semihosting (formerly SWI)
        AREA  BlockData, DATA, READWRITE
src     DCD   1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst     DCD   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        END
```

You can make this module more efficient by using `LDM` and `STM` for as much of the copying as possible. Eight is a sensible number of words to transfer at a time, given the number of available registers. You can find the number of eight-word multiples in the block to be copied (if `R2` = number of words to be copied) using:

```
    MOVS   r3, r2, LSR #3    ; number of eight word multiples
```

You can use this value to control the number of iterations through a loop that copies eight words per iteration. When there are fewer than eight words left, you can find the number of words left (assuming that `R2` has not been corrupted) using:

```
    ANDS   r2, r2, #7
```

### Example of block copy using LDM and STM

The following example lists the block copy module rewritten to use `LDM` and `STM` for copying:

```
      AREA   Block, CODE, READONLY ; name this block of code
num   EQU    20                    ; set number of words to be copied
      ENTRY                        ; mark the first instruction called
start
      LDR    r0, =src              ; r0 = pointer to source block
      LDR    r1, =dst              ; r1 = pointer to destination block
      MOV    r2, #num              ; r2 = number of words to copy
      MOV    sp, #0x400            ; Set up stack pointer (sp)
blockcopy
      MOVS   r3,r2, LSR #3         ; Number of eight word multiples
      BEQ    copywords             ; Fewer than eight words to move?
      PUSH   {r4-r11}              ; Save some working registers
octcopy
      LDM    r0!, {r4-r11}         ; Load 8 words from the source
      STM    r1!, {r4-r11}         ; and put them at the destination
      SUBS   r3, r3, #1            ; Decrement the counter
      BNE    octcopy               ; ... copy more
      POP    {r4-r11}              ; Don't require these now - restore
                                   ; originals
copywords
      ANDS   r2, r2, #7            ; Number of odd words to copy
      BEQ    stop                  ; No words left to copy?
wordcopy
      LDR    r3, [r0], #4          ; Load a word from the source and
      STR    r3, [r1], #4          ; store it to the destination
      SUBS   r2, r2, #1            ; Decrement the counter
      BNE    wordcopy              ; ... copy more
stop
      MOV    r0, #0x18             ; angel_SWIreason_ReportException
      LDR    r1, =0x20026          ; ADP_Stopped_ApplicationExit
        SVC   #0x123456             ; AArch32 semihosting (formerly SWI)
      AREA   BlockData, DATA, READWRITE
src   DCD    1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst   DCD    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      END
```

———— **Note** ————

The purpose of this example is to show the use of the `LDM` and `STM` instructions. There are other ways to perform bulk copy operations, the most efficient of which depends on many factors and is outside the scope of this document.

————————————

***Related information***

*What is the fastest way to copy memory on a Cortex-A8?*

## 3.19 Memory accesses

Many load and store instructions support different addressing modes.

**Offset addressing**

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The base register is unchanged. The assembly language syntax for this mode is:

```
[Rn, offset]
```

**Pre-indexed addressing**

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register. The assembly language syntax for this mode is:

```
[Rn, offset]!
```

**Post-indexed addressing**

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register. The assembly language syntax for this mode is:

```
[Rn], offset
```

In each case, `Rn` is the base register and `offset` can be:

- An immediate constant.
- An index register, `Rm`.
- A shifted index register, such as `Rm`, `LSL #shift`.

***Related concepts***

*4.15 Address alignment in A32/T32 code* on page 4-91

## 3.20 The Read-Modify-Write operation

The read-modify-write operation ensures that you modify only the specific bits in a system register that you want to change.

Individual bits in a system register control different system functionality. Modifying the wrong bits in a system register might cause your program to behave incorrectly.

```
    VMRS    r10,FPSCR              ; copy FPSCR into the general-purpose r10
    BIC     r10,r10,#0x00370000    ; clear STRIDE bits[21:20] and LEN bits[18:16]
    ORR     r10,r10,#0x00030000    ; set bits[17:16] (STRIDE =1 and LEN = 4)
    VMSR    FPSCR,r10              ; copy r10 back into FPSCR
```

To read-modify-write a system register, the instruction sequence is:

1. The first instruction copies the value from the target system register to a temporary general-purpose register.
2. The next one or more instructions modify the required bits in the general-purpose register. This can be one or both of:
   - `BIC` to clear to 0 only the bits that must be cleared.
   - `ORR` to set to 1 only the bits that must be set.
3. The final instruction writes the value from the general-purpose register to the target system register.

## 3.21 Optional hash with immediate constants

You do not have to specify a hash before an immediate constant in any instruction syntax.

This applies to A32, T32, Advanced SIMD, and floating-point instructions. For example, the following are valid instructions:

```
    BKPT 100
    MOVT R1, 256
    VCEQ.I8 Q1, Q2, 0
```

By default, the assembler warns if you do not specify a hash:

```
WARNING: A1865W: '#' not seen before constant expression.
```

You can suppressed this with `--diag_suppress=1865`.

If you use the assembly code with another assembler, you are advised to use the # before all immediates. The disassembler always shows the # for clarity.

## 3.22 Use of macros

A macro definition is a block of code enclosed between `MACRO` and `MEND` directives. It defines a name that you can use as a convenient alternative to repeating the block of code.

The main uses for a macro are:

- To make it easier to follow the logic of the source code by replacing a block of code with a single meaningful name.
- To avoid repeating a block of code several times.

***Related concepts***

*3.23 Test-and-branch macro example* on page 3-64

*3.24 Unsigned integer division macro example* on page 3-65

***Related reference***

*7.51 MACRO and MEND* on page 7-256

## 3.23 Test-and-branch macro example

You can use a macro to perform a test-and-branch operation.

In A32 code, a test-and-branch operation requires two instructions to implement.

You can define a macro such as this:

```
        MACRO
$label  TestAndBranch  $dest, $reg, $cc
$label  CMP     $reg, #0
        B$cc    $dest
        MEND
```

The line after the `MACRO` directive is the *macro prototype statement*. This defines the name (`TestAndBranch`) you use to invoke the macro. It also defines parameters (`$label`, `$dest`, `$reg`, and `$cc`). Unspecified parameters are substituted with an empty string. For this macro you must give values for `$dest`, `$reg` and `$cc` to avoid syntax errors. The assembler substitutes the values you give into the code.

This macro can be invoked as follows:

```
test    TestAndBranch    NonZero, r0, NE
         ...
         ...
NonZero
```

After substitution this becomes:

```
test    CMP     r0, #0
        BNE     NonZero
         ...
         ...
NonZero
```

***Related concepts***

*3.22 Use of macros* on page 3-63

*3.24 Unsigned integer division macro example* on page 3-65

*6.10 Numeric local labels* on page 6-177

## 3.24 Unsigned integer division macro example

You can use a macro to perform unsigned integer division.

The macro takes the following parameters:

**$Bot**

The register that holds the divisor.

**$Top**

The register that holds the dividend before the instructions are executed. After the instructions are executed, it holds the remainder.

**$Div**

The register where the quotient of the division is placed. It can be `NULL` (`""`) if only the remainder is required.

**$Temp**

A temporary register used during the calculation.

### Example unsigned integer division with a macro

```
        MACRO
$Lab    DivMod  $Div,$Top,$Bot,$Temp
        ASSERT  $Top <> $Bot        ; Produce an error message if the
        ASSERT  $Top <> $Temp       ; registers supplied are
        ASSERT  $Bot <> $Temp       ; not all different
        IF      "$Div" <> ""
            ASSERT  $Div <> $Top    ; These three only matter if $Div
            ASSERT  $Div <> $Bot    ; is not null ("")
            ASSERT  $Div <> $Temp   ;
        ENDIF
$Lab
        MOV     $Temp, $Bot           ; Put divisor in $Temp
        CMP     $Temp, $Top, LSR #1   ; double it until
90      MOVLS   $Temp, $Temp, LSL #1  ; 2 * $Temp > $Top
        CMP     $Temp, $Top, LSR #1
        BLS     %b90                ; The b means search backwards
        IF      "$Div" <> ""        ; Omit next instruction if $Div
                                    ; is null
            MOV     $Div, #0        ; Initialize quotient
        ENDIF
91      CMP     $Top, $Temp         ; Can we subtract $Temp?
        SUBCS   $Top, $Top,$Temp    ; If we can, do so
        IF      "$Div" <> ""        ; Omit next instruction if $Div
                                    ; is null
            ADC     $Div, $Div, $Div  ; Double $Div
        ENDIF
        MOV     $Temp, $Temp, LSR #1  ; Halve $Temp,
        CMP     $Temp, $Bot           ; and loop until
        BHS     %b91                  ; less than divisor
        MEND
```

The macro checks that no two parameters use the same register. It also optimizes the code produced if only the remainder is required.

To avoid multiple definitions of labels if `DivMod` is used more than once in the assembler source, the macro uses numeric local labels (90, 91).

The following example shows the code that this macro produces if it is invoked as follows:

```
ratio  DivMod  R0,R5,R4,R2
```

### Output from the example division macro

```
        ASSERT  r5 <> r4          ; Produce an error if the
        ASSERT  r5 <> r2          ; registers supplied are
        ASSERT  r4 <> r2          ; not all different
        ASSERT  r0 <> r5          ; These three only matter if $Div
        ASSERT  r0 <> r4          ; is not null ("")
        ASSERT  r0 <> r2            ;
ratio
        MOV     r2, r4              ; Put divisor in $Temp
        CMP     r2, r5, LSR #1      ; double it until
90      MOVLS   r2, r2, LSL #1      ; 2 * r2 > r5
        CMP     r2, r5, LSR #1
        BLS     %b90                ; The b means search backwards
        MOV     r0, #0              ; Initialize quotient
91      CMP     r5, r2              ; Can we subtract r2?
        SUBCS   r5, r5, r2          ; If we can, do so
        ADC     r0, r0, r0          ; Double r0
        MOV     r2, r2, LSR #1      ; Halve r2,
        CMP     r2, r4              ; and loop until
        BHS     %b91                ; less than divisor
```

***Related concepts***

*3.22 Use of macros* on page 3-63

*3.23 Test-and-branch macro example* on page 3-64

*6.10 Numeric local labels* on page 6-177

## 3.25 Instruction and directive relocations

The assembler can embed relocation directives in object files to indicate labels with addresses that are unknown at assembly time. The assembler can relocate several types of instruction.

A relocation is a directive embedded in the object file that enables source code to refer to a label whose target address is unknown or cannot be calculated at assembly time. The assembler emits a relocation in the object file, and the linker resolves this to the address where the target is placed.

The assembler relocates the data directives `DCB`, `DCW`, `DCWU`, `DCD`, and `DCDU` if their syntax contains an external symbol, that is a symbol declared using `IMPORT` or `EXTERN`. This causes the bottom 8, 16, or 32 bits of the address to be used at link-time.

The `REQUIRE` directive emits a relocation to signal to the linker that the target label must be present if the current section is present.

The assembler is permitted to emit a relocation for these instructions:

**`LDR` (PC-relative)**
All A32 and T32 instructions, except the T32 doubleword instruction, can be relocated.

**`PLD`, `PLDW`, and `PLI`**
All A32 and T32 instructions can be relocated.

**`B`, `BL`, and `BLX`**
All A32 and T32 instructions can be relocated.

**`CBZ` and `CBNZ`**
All T32 instructions can be relocated but this is discouraged because of the limited branch range of these instructions.

**`LDC` and `LDC2`**
Only A32 instructions can be relocated.

**`VLDR`**
Only A32 instructions can be relocated.

The assembler emits a relocation for these instructions if the label used meets any of the following requirements, as appropriate for the instruction type:

- The label is `WEAK`.
- The label is not in the same `AREA`.
- The label is external to the object (`IMPORT` or `EXTERN`).

For `B`, `BL`, and `BX` instructions, the assembler emits a relocation also if:

- The label is a function.
- The label is exported using `EXPORT` or `GLOBAL`.

**Note**

You can use the `RELOC` directive to control the relocation at a finer level, but this requires knowledge of the ABI.

### Example

```
    IMPORT sym   ; sym is an external symbol
    DCW sym      ; Because DCW only outputs 16 bits, only the lower
                 ; 16 bits of the address of sym are inserted at
                 ; link-time.
```

***Related reference***

*7.6 AREA* on page 7-205

*7.27 EXPORT or GLOBAL* on page 7-229

*7.45 IMPORT and EXTERN* on page 7-249
*7.58 REQUIRE* on page 7-267
*7.57 RELOC* on page 7-266
*7.15 DCB* on page 7-217
*7.16 DCD and DCDU* on page 7-218
*7.22 DCW and DCWU* on page 7-224
***Related information***
*ELF for the Arm Architecture*

## 3.26 Symbol versions

The Arm linker conforms to the Base Platform ABI for the Arm Architecture (BPABI) and supports the GNU-extended symbol versioning model.

To add a symbol version to an existing symbol, you must define a version symbol at the same address. A version symbol is of the form:

- *`name@ver`* if *`ver`* is a non default version of *`name`*.
- *`name@@ver`* if *`ver`* is the default version of *`name`*.

The version symbols must be enclosed in vertical bars.

For example, to define a default version:

```
|my_versioned_symbol@@ver2|   ; Default version
my_asm_function PROC
                ...
                BX lr
                ENDP
```

To define a non default version:

```
|my_versioned_symbol@ver1|    ; Non default version
my_old_asm_function     PROC
                        ...
                        BX lr
                        ENDP
```

***Related information***

*Base Platform ABI for the Arm Architecture*

*Accessing and managing symbols with armlink*

## 3.27 Frame directives

Frame directives provide information in object files that enables debugging and profiling of assembly language functions.

You must use frame directives to describe the way that your code uses the stack if you want to be able to do either of the following:

- Debug your application using stack unwinding.
- Use either flat or call-graph profiling.

The assembler uses frame directives to insert DWARF debug frame information into the object file in ELF format that it produces. This information is required by a debugger for stack unwinding and for profiling.

Be aware of the following:

- Frame directives do not affect the code produced by the assembler.
- The assembler does not validate the information in frame directives against the instructions emitted.

***Related concepts***

*3.28 Exception tables and Unwind tables* on page 3-71

***Related reference***

*7.3 About frame directives* on page 7-201

***Related information***

*Procedure Call Standard for the Arm Architecture*

## 3.28 Exception tables and Unwind tables

You use `FRAME` directives to enable the assembler to generate *unwind* tables.

**Note**

Not supported for AArch64 state.

Exception tables are necessary to handle exceptions thrown by functions in high-level languages such as C++. Unwind tables contain debug frame information which is also necessary for the handling of such exceptions. An exception can only propagate through a function with an unwind table.

An assembly language function is code enclosed by either `PROC` and `ENDP` or `FUNC` and `ENDFUNC` directives. Functions written in C++ have unwind information by default. However, for assembly language functions that are called from C++ code, you must ensure that there are exception tables and unwind tables to enable the exceptions to propagate through them.

An exception cannot propagate through a function with a *nounwind* table. The exception handling runtime environment terminates the program if it encounters a nounwind table during exception processing.

The assembler can generate nounwind table entries for all functions and non-functions. The assembler can generate an unwind table for a function only if the function contains sufficient `FRAME` directives to describe the use of the stack within the function. To be able to create an unwind table for a function, each `POP` or `PUSH` instruction must be followed by a `FRAME POP` or `FRAME PUSH` directive respectively. Functions must conform to the conditions set out in the *Exception Handling ABI for the Arm® Architecture* (EHABI), section 9.1 *Constraints on Use*. If the assembler cannot generate an unwind table it generates a nounwind table.

***Related concepts***

*3.27 Frame directives* on page 3-70

***Related reference***

*7.3 About frame directives* on page 7-201

*5.26 --exceptions, --no_exceptions* on page 5-124

*5.27 --exceptions_unwind, --no_exceptions_unwind* on page 5-125

*7.39 FRAME UNWIND ON* on page 7-242

*7.40 FRAME UNWIND OFF* on page 7-243

*7.41 FUNCTION or PROC* on page 7-244

*7.24 ENDFUNC or ENDP* on page 7-226

***Related information***

*Exception Handling ABI for the Arm Architecture*

# Chapter 4
# Using armasm

Describes how to use `armasm`.

It contains the following sections:

## 4.1 armasm command-line syntax

You can use a command line to invoke `armasm`. You must specify an input source file and you can specify various options.

The command for invoking the assembler is:

```
armasm {options} inputfile
```

where:

***options***

are commands that instruct the assembler how to assemble the *inputfile*. You can invoke `armasm` with any combination of options separated by spaces. You can specify values for some options. To specify a value for an option, use either '`=`' (*option=value*) or a space character (*option value*).

***inputfile***

is an assembly source file. It must contain UAL, pre-UAL A32 or T32, or A64 assembly language.

The assembler command line is case-insensitive, except in filenames and where specified. The assembler uses the same command-line ordering rules as the compiler. This means that if the command line contains options that conflict with each other, then the last option found always takes precedence.

## 4.2 Specify command-line options with an environment variable

The `ARMCOMPILER6_ASMOPT` environment variable can hold command-line options for the assembler.

The syntax is identical to the command-line syntax. The assembler reads the value of `ARMCOMPILER6_ASMOPT` and inserts it at the front of the command string. This means that options specified in `ARMCOMPILER6_ASMOPT` can be overridden by arguments on the command line.

***Related concepts***

*4.1 armasm command-line syntax* on page 4-73

***Related information***

*Toolchain environment variables*

## 4.3 Using stdin to input source code to the assembler

You can use stdin to pipe output from another program into armasm or to input source code directly on the command line. This is useful if you want to test a short piece of code without having to create a file for it.

To use `stdin` to pipe output from another program into `armasm`, invoke the program and the assembler using the pipe character (|). Use the minus character (`-`) as the source filename to instruct the assembler to take input from `stdin`. You must specify the output filename using the `-o` option. You can specify the command-line options you want to use. For example to pipe output from `fromelf`:

```
fromelf --disassemble A32input.o | armasm --cpu=8-A.32 -o A32output.o -
```

**Note**

The source code from `stdin` is stored in an internal cache that can hold up to 8 MB. You can increase this cache size using the `--maxcache` command-line option.

To use `stdin` to input source code directly on the command line:

### Procedure

1. Invoke the assembler with the command-line options you want to use. Use the minus character (`-`) as the source filename to instruct the assembler to take input from `stdin`. You must specify the output filename using the `-o` option. For example:

   ```
   armasm --cpu=8-A.32 -o output.o -
   ```

2. Enter your input. For example:

   ```
           AREA     A32ex, CODE, READONLY
                                  ; Name this block of code A32ex
           ENTRY                  ; Mark first instruction to execute
   start
           MOV      r0, #10       ; Set up parameters
           MOV      r1, #3
           ADD      r0, r0, r1    ; r0 = r0 + r1
   stop
           MOV      r0, #0x18     ; angel_SWIreason_ReportException
           LDR      r1, =0x20026  ; ADP_Stopped_ApplicationExit
           SVC      #0x123456     ; AArch32 semihosting (formerly SWI)

           END                    ; Mark end of file
   ```

3. Terminate your input by entering:
   - `Ctrl+Z` then `Return` on Microsoft Windows systems.
   - `Ctrl+D` on Unix-based operating systems.

***Related concepts***

*4.1 armasm command-line syntax* on page 4-73

***Related reference***

*5.44 --maxcache=n* on page 5-142

## 4.4 Built-in variables and constants

`armasm` defines built-in variables that hold information about, for example, the state of `armasm`, the command-line options used, and the target architecture or processor.

The following table lists the built-in variables defined by `armasm`:

**Table 4-1 Built-in variables**

| | |
|---|---|
| `{ARCHITECTURE}` | Holds the name of the selected Arm architecture. |
| `{AREANAME}` | Holds the name of the current `AREA`. |
| `{ARMASM_VERSION}` | Holds an integer that increases with each version of `armasm`. The format of the version number is *`Mmmuuxx`* where:<br>• *`M`* is the major version number, `6`.<br>• *`mm`* is the minor version number.<br>• *`uu`* is the update number.<br>• *`xx`* is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.<br>**Note**<br>The built-in variable `\|ads$version\|` is deprecated. |
| `\|ads$version\|` | Has the same value as `{ARMASM_VERSION}`. |
| `{CODESIZE}` | Is a synonym for `{CONFIG}`. |
| `{COMMANDLINE}` | Holds the contents of the command line. |
| `{CONFIG}` | Has the value:<br>• `64` if the assembler is assembling A64 code.<br>• `32` if the assembler is assembling A32 code.<br>• `16` if the assembler is assembling T32 code. |
| `{CPU}` | Holds the name of the selected processor. The value of `{CPU}` is derived from the value specified in the `--cpu` option on the command line. |
| `{ENDIAN}` | Has the value `"big"` if the assembler is in big-endian mode, or `"little"` if it is in little-endian mode. |
| `{FPU}` | Holds the name of the selected FPU. The default in AArch32 state is `"FP-ARMv8"`. The default in AArch64 state is `"A64"`. |
| `{INPUTFILE}` | Holds the name of the current source file. |
| `{INTER}` | Has the Boolean value True if `--apcs=/inter` is set. The default is `{False}`. |
| `{LINENUM}` | Holds an integer indicating the line number in the current source file. |
| `{LINENUMUP}` | When used in a macro, holds an integer indicating the line number of the current macro. The value is the same as `{LINENUM}` when used in a non-macro context. |
| `{LINENUMUPPER}` | When used in a macro, holds an integer indicating the line number of the top macro. The value is the same as `{LINENUM}` when used in a non-macro context. |
| `{OPT}` | Value of the currently-set listing option. You can use the `OPT` directive to save the current listing option, force a change in it, or restore its original value. |

| | |
|---|---|
| `{PC}` or `.` | Address of current instruction. |
| `{PCSTOREOFFSET}` | Is the offset between the address of the `STR PC,[…]` or `STM Rb,{…, PC}` instruction and the value of PC stored out. This varies depending on the processor or architecture specified. |
| `{ROPI}` | Has the Boolean value `{True}` if `--apcs=/ropi` is set. The default is `{False}`. |
| `{RWPI}` | Has the Boolean value `{True}` if `--apcs=/rwpi` is set. The default is `{False}`. |
| `{VAR}` or `@` | Current value of the storage area location counter. |

You can use built-in variables in expressions or conditions in assembly source code. For example:

```
        IF {ARCHITECTURE} = "8-A"
```

They cannot be set using the `SETA`, `SETL`, or `SETS` directives.

The names of the built-in variables can be in uppercase, lowercase, or mixed, for example:

```
        IF {CpU} = "Generic ARM"
```

**Note**

All built-in string variables contain case-sensitive values. Relational operations on these built-in variables do not match with strings that contain an incorrect case. Use the command-line options `--cpu` and `--fpu` to determine valid values for `{CPU}`, `{ARCHITECTURE}`, and `{FPU}`.

The assembler defines the built-in Boolean constants `TRUE` and `FALSE`.

**Table 4-2 Built-in Boolean constants**

| | |
|---|---|
| `{FALSE}` | Logical constant false. |
| `{TRUE}` | Logical constant true. |

The following table lists the target processor-related built-in variables that are predefined by the assembler. Where the value field is empty, the symbol is a Boolean value and the meaning column describes when its value is `{TRUE}`.

**Table 4-3 Predefined macros**

| Name | Value | Meaning |
|---|---|---|
| `{TARGET_ARCH_AARCH32}` | `boolean` | `{TRUE}` when assembling for AArch32 state. `{FALSE}` when assembling for AArch64 state. |
| `{TARGET_ARCH_AARCH64}` | `boolean` | `{TRUE}` when assembling for AArch64 state. `{FALSE}` when assembling for AArch32 state. |
| `{TARGET_ARCH_ARM}` | *`num`* | The number of the A32 base architecture of the target processor irrespective of whether the assembler is assembling for A32 or T32. The value is defined as zero when assembling for A64, and eight when assembling for A32/T32. |
| `{TARGET_ARCH_THUMB}` | *`num`* | The number of the T32 base architecture of the target processor irrespective of whether the assembler is assembling for A32 or T32. The value is defined as zero when assembling for A64, and five when assembling for A32/T32. |

**Table 4-3 Predefined macros (continued)**

| Name | Value | Meaning |
|---|---|---|
| `{TARGET_ARCH_XX}` | – | *XX* represents the target architecture and its value depends on the target processor:<br>For the Armv8 architecture:<br>• If you specify the assembler option `--cpu=8-A.32` or `--cpu=8-A.64` then `{TARGET_ARCH_8_A}` is defined.<br>• If you specify the assembler option `--cpu=8.1-A.32` or `--cpu=8.1-A.64` then `{TARGET_ARCH_8_1_A}` is defined.<br>For the Armv7 architecture, if you specify `--cpu=Cortex-A8`, for example, then `{TARGET_ARCH_7_A}` is defined. |
| `{TARGET_FEATURE_EXTENSION_REGISTER_COUNT}` | *num* | The number of 64-bit extension registers available in Advanced SIMD or floating-point. |
| `{TARGET_FEATURE_CLZ}` | – | If the target processor supports the `CLZ` instruction. |
| `{TARGET_FEATURE_CRYPTOGRAPHY}` | – | If the target processor has cryptographic instructions. |
| `{TARGET_FEATURE_DIVIDE}` | – | If the target processor supports the hardware divide instructions `SDIV` and `UDIV`. |
| `{TARGET_FEATURE_DOUBLEWORD}` | – | If the target processor supports doubleword load and store instructions, for example the A32 and T32 instructions `LDRD` and `STRD` (except the Armv6-M architecture). |
| `{TARGET_FEATURE_DSPMUL}` | – | If the DSP-enhanced multiplier (for example the `SMLAxy` instruction) is available. |
| `{TARGET_FEATURE_MULTIPLY}` | – | If the target processor supports long multiply instructions, for example the A32 and T32 instructions `SMULL`, `SMLAL`, `UMULL`, and `UMLAL` (that is, all architectures except the Armv6-M architecture). |
| `{TARGET_FEATURE_MULTIPROCESSING}` | – | If assembling for a target processor with Multiprocessing Extensions. |
| `{TARGET_FEATURE_NEON}` | – | If the target processor has Advanced SIMD. |
| `{TARGET_FEATURE_NEON_FP16}` | – | If the target processor has Advanced SIMD with half-precision floating-point operations. |
| `{TARGET_FEATURE_NEON_FP32}` | – | If the target processor has Advanced SIMD with single-precision floating-point operations. |
| `{TARGET_FEATURE_NEON_INTEGER}` | – | If the target processor has Advanced SIMD with integer operations. |
| `{TARGET_FEATURE_UNALIGNED}` | – | If the target processor has support for unaligned accesses (all architectures except the Armv6-M architecture). |
| `{TARGET_FPU_SOFTVFP}` | – | If assembling with the option `--fpu=SoftVFP`. |
| `{TARGET_FPU_SOFTVFP_VFP}` | – | If assembling for a target processor with SoftVFP and floating-point hardware, for example `--fpu=SoftVFP+FP-ARMv8`. |

**Table 4-3 Predefined macros (continued)**

| Name | Value | Meaning |
|---|---|---|
| `{TARGET_FPU_VFP}` | – | If assembling for a target processor with floating-point hardware, without using SoftVFP, for example `--fpu=FP-ARMv8`. |
| `{TARGET_FPU_VFPV2}` | – | If assembling for a target processor with VFPv2. |
| `{TARGET_FPU_VFPV3}` | – | If assembling for a target processor with VFPv3. |
| `{TARGET_FPU_VFPV4}` | – | If assembling for a target processor with VFPv4. |
| `{TARGET_PROFILE_A}` | – | If assembling for a Cortex®-A profile processor, for example, if you specify the assembler option `--cpu=7-A`. |
| `{TARGET_PROFILE_M}` | – | If assembling for a Cortex-M profile processor, for example, if you specify the assembler option `--cpu=7-M`. |
| `{TARGET_PROFILE_R}` | – | If assembling for a Cortex-R profile processor, for example, if you specify the assembler option `--cpu=7-R`. |

***Related concepts***

*4.5 Identifying versions of armasm in source code* on page 4-80

***Related reference***

*5.13 --cpu=name* on page 5-109

*5.32 --fpu=name* on page 5-130

## 4.5 Identifying versions of armasm in source code

The assembler defines the built-in variable `ARMASM_VERSION` to hold the version number of the assembler.

You can use it as follows:

```
    IF ( {ARMASM_VERSION} / 100000) >= 6
        ; using armasm in Arm Compiler 6
    ELIF ( {ARMASM_VERSION} / 1000000) = 5
        ; using armasm in Arm Compiler 5
    ELSE
        ; using armasm in Arm Compiler 4.1 or earlier
    ENDIF
```

**Note**

The built-in variable `|ads$version|` is deprecated.

***Related reference***

*4.4 Built-in variables and constants* on page 4-76

## 4.6 Diagnostic messages

The assembler can provide extra error, warning, and remark diagnostic messages in addition to the default ones.

By default, these additional diagnostic messages are not displayed. However, you can enable them using the command-line options `--diag_error`, `--diag_warning`, and `--diag_remark`.

***Related concepts***

*4.7 Interlocks diagnostics* on page 4-82
*4.8 Automatic IT block generation in T32 code* on page 4-83
*4.9 T32 branch target alignment* on page 4-84
*4.10 T32 code size diagnostics* on page 4-85
*4.11 A32 and T32 instruction portability diagnostics* on page 4-86
*4.12 T32 instruction width diagnostics* on page 4-87
*4.13 Two pass assembler diagnostics* on page 4-88

***Related reference***

*5.17 --diag_error=tag[,tag,...]* on page 5-115

## 4.7 Interlocks diagnostics

`armasm` can report warning messages about possible interlocks in your code caused by the pipeline of the processor chosen by the `--cpu` option.

To do this, use the `--diag_warning 1563` command-line option when invoking `armasm`.

**Note**

- `armasm` does not have an accurate model of the target processor, so these messages are not reliable when used with a multi-issue processor such as Cortex-A8.
- Interlocks diagnostics apply to A32 and T32 code, but not to A64 code.

***Related concepts***

*4.8 Automatic IT block generation in T32 code* on page 4-83

*4.9 T32 branch target alignment* on page 4-84

*4.12 T32 instruction width diagnostics* on page 4-87

*4.6 Diagnostic messages* on page 4-81

***Related reference***

*5.21 --diag_warning=tag[,tag,…]* on page 5-119

## 4.8 Automatic IT block generation in T32 code

`armasm` can automatically insert an `IT` block for conditional instructions in T32 code, without requiring the use of explicit `IT` instructions.

If you write the following code:

```
        AREA x, CODE
        THUMB
        MOVNE   r0,r1
        NOP
        IT      NE
        MOVNE   r0,r1
        END
```

`armasm` generates the following instructions:

```
        IT      NE
        MOVNE   r0,r1
        NOP
        IT      NE
        MOVNE   r0,r1
```

You can receive warning messages about the automatic generation of `IT` blocks when assembling T32 code. To do this, use the `armasm --diag_warning 1763` command-line option when invoking `armasm`.

***Related concepts***

*4.6 Diagnostic messages* on page 4-81

***Related reference***

*5.21 --diag_warning=tag[,tag,…]* on page 5-119

## 4.9 T32 branch target alignment

`armasm` can issue warnings about non word-aligned branch targets in T32 code.

On some processors, non word-aligned T32 instructions sometimes take one or more additional cycles to execute in loops. This means that it can be an advantage to ensure that branch targets are word-aligned. To ensure `armasm` reports such warnings, use the `--diag_warning 1604` command-line option when invoking it.

***Related concepts***

*4.6 Diagnostic messages* on page 4-81

***Related reference***

*5.21 --diag_warning=tag[,tag,…]* on page 5-119

## 4.10 T32 code size diagnostics

In T32 code, some instructions, for example a branch or `LDR` (PC-relative), can be encoded as either a 32-bit or 16-bit instruction. `armasm` chooses the size of the instruction encoding.

`armasm` can issue a warning when it assembles a T32 instruction to a 32-bit encoding when it could have used a 16-bit encoding.

To enable this warning, use the `--diag_warning 1813` command-line option when invoking `armasm`.

***Related concepts***

*4.17 Instruction width selection in T32 code* on page 4-93

*4.6 Diagnostic messages* on page 4-81

***Related reference***

*5.21 --diag_warning=tag[,tag,…]* on page 5-119

## 4.11 A32 and T32 instruction portability diagnostics

`armasm` can issue warnings about instructions that cannot assemble to both A32 and T32 code.

There are a few UAL instructions that can assemble as either A32 code or T32 code, but not both. You can identify these instructions in the source code using the `--diag_warning 1812` command-line option when invoking `armasm`.

It warns for any instruction that cannot be assembled in the other instruction set. This is only a hint, and other factors, like relocation availability or target distance might affect the accuracy of the message.

***Related concepts***

*4.6 Diagnostic messages* on page 4-81

***Related reference***

*5.21 --diag_warning=tag[,tag,…]* on page 5-119

## 4.12 T32 instruction width diagnostics

`armasm` can issue a warning when it assembles a T32 instruction to a 32-bit encoding when it could have used a 16-bit encoding.

If you use the `.W` specifier, the instruction is encoded in 32 bits even if it could be encoded in 16 bits. You can use a diagnostic warning to detect when a branch instruction could have been encoded in 16 bits, but has been encoded in 32 bits. To do this, use the `--diag_warning 1607` command-line option when invoking `armasm`.

——— **Note** ———

This diagnostic does not produce a warning for relocated branch instructions, because the final address is not known. The linker might even insert a veneer, if the branch is out of range for a 32-bit instruction.

———————

***Related concepts***

*4.6 Diagnostic messages* on page 4-81

***Related reference***

*5.21 --diag_warning=tag[,tag,...]* on page 5-119

## 4.13 Two pass assembler diagnostics

`armasm` can issue a warning about code that might not be identical in both assembler passes.

`armasm` is a two pass assembler and the input code that the assembler reads must be identical in both passes. If a symbol is defined after the `:DEF:` test for that symbol, then the code read in pass one might be different from the code read in pass two. `armasm` can warn in this situation.

To do this, use the `--diag_warning 1907` command-line option when invoking `armasm`.

### Example

The following example shows that the symbol `foo` is defined after the `:DEF: foo` test.

```
    AREA x,CODE
    [ :DEF: foo
    ]
foo MOV r3, r4
    END
```

Assembling this code with `--diag_warning 1907` generates the message:

```
Warning A1907W: Test for this symbol has been seen and may cause failure in the second pass.
```

***Related concepts***

*4.8 Automatic IT block generation in T32 code* on page 4-83

*4.9 T32 branch target alignment* on page 4-84

*4.12 T32 instruction width diagnostics* on page 4-87

*4.6 Diagnostic messages* on page 4-81

*1.3 How the assembler works* on page 1-19

***Related reference***

*5.21 --diag_warning=tag[,tag,…]* on page 5-119

*1.4 Directives that can be omitted in pass 2 of the assembler* on page 1-21

## 4.14 Using the C preprocessor

`armasm` can invoke `armclang` to preprocess an assembly language source file before assembling it. This allows you to use C preprocessor commands in assembly source code.

If you do this, you must use the `--cpreproc` command-line option together with the `--cpreproc_opts` command-line option when invoking the assembler. This causes `armasm` to call `armclang` to preprocess the file before assembling it.

**Note**

As a minimum, you must specify the `armclang` `--target` option and either the `-mcpu` or `-march` option with `--cpreproc_opts`.

---

`armasm` looks for the `armclang` binary in the same directory as the `armasm` binary. If it does not find the binary, it expects it to be on the PATH.

`armasm` passes the following options by default to `armclang` if present on the command line:
- Basic pre-processor configuration options, such as `-E`.
- User specified include directories, `-I` directives.
- User specified licensing options, such as `--site_license`.
- Anything specified in `--cpreproc_opts`.

Some of the options that `armasm` passes to `armclang` are converted to the `armclang` equivalent beforehand. These are shown in the following table:

**Table 4-4 armclang equivalent command-line options**

| armasm | armclang |
|---|---|
| `--thumb` | `-mthumb` |
| `--arm` | `-marm` |
| `-i` | `-I` |

`armasm` correctly interprets the preprocessed `#line` commands. It can generate error messages and `debug_line` tables using the information in the `#line` commands.

### Preprocessing an assembly language source file

The following example shows the command you write to preprocess and assemble a file, `source.S`. The example also passes the compiler options to define a macro called `RELEASE`, and to undefine a macro called `ALPHA`.

```
armasm --cpu=cortex-m3 --cpreproc --cpreproc_opts=--target=arm-arm-none-eabi,-mcpu=cortex-
a9,-D,RELEASE,-U,ALPHA source.S
```

### Preprocessing an assembly language source file manually

Alternatively, you must manually call `armclang` to preprocess the file before calling `armasm`. The following example shows the commands you write to manually preprocess and assemble a file, `source.S`:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -E source.S > preprocessed.S
armasm --cpu=cortex-m3 preprocessed.S
```

In this example, the preprocessor outputs a file called `preprocessed.S`, and `armasm` assembles it.

***Related reference***

*5.10 --cpreproc* on page 5-106

*5.11 --cpreproc_opts=option[,option,…]* on page 5-107

***Related information***

*Specifying a target architecture, processor, and instruction set*

*-march armclang option*

*-mcpu armclang option*

*--target armclang option*

## 4.15 Address alignment in A32/T32 code

In Armv7-A and Armv7-R, the A bit in the *System Control Register* (SCTLR) controls whether alignment checking is enabled or disabled. In Armv7-M, the `UNALIGN_TRP` bit, bit 3, in the *Configuration and Control Register* (CCR) controls this.

If alignment checking is enabled, all unaligned word and halfword transfers cause an alignment exception. If disabled, unaligned accesses are permitted for the `LDR`, `LDRH`, `STR`, `STRH`, `LDRSH`, `LDRT`, `STRT`, `LDRSHT`, `LDRHT`, `STRHT`, and `TBH` instructions. Other data-accessing instructions always cause an alignment exception for unaligned data.

For `STRD` and `LDRD`, the specified address must be word-aligned.

If all your data accesses are aligned, you can use the `--no_unaligned_access` command-line option to declare that the output object was not permitted to make unaligned access. The linker can then avoid linking in any library functions that support unaligned access if all input objects declare that they were not permitted to use unaligned accesses.

***Related reference***

*5.60 --unaligned_access, --no_unaligned_access* on page 5-158

## 4.16 Address alignment in A64 code

If alignment checking is not enabled, then unaligned accesses are permitted for all load and store instructions other than exclusive load, exclusive store, load acquire, and store release instructions. If alignment checking is enabled, then unaligned accesses are not permitted.

This means all load and store instructions must use addresses that are aligned to the size of the data being accessed. In other words, addresses for 8-byte transfers must be 8-byte aligned, addresses for 4-byte transfers are 4-byte word aligned, and addresses for 2-byte transfers are 2-byte aligned. Unaligned accesses cause an alignment exception.

For any memory access, if the stack pointer is used as the base register, then it must be quadword aligned. Otherwise it generates a stack alignment exception.

If all your data accesses are aligned, you can use the `--no_unaligned_access` command-line option to declare that the output object was not permitted to make unaligned access. The linker can then avoid linking in any library functions that support unaligned access if all input objects declare that they were not permitted to use unaligned accesses.

## 4.17 Instruction width selection in T32 code

Some T32 instructions can have either a 16-bit encoding or a 32-bit encoding.

If you do not specify the instruction size, by default:

- For forward reference `LDR`, `ADR`, and `B` instructions, `armasm` always generates a 16-bit instruction, even if that results in failure for a target that could be reached using a 32-bit instruction.
- For external reference `LDR` and `B` instructions, `armasm` always generates a 32-bit instruction.
- In all other cases, `armasm` generates the smallest size encoding that can be output.

If you want to override this behavior, you can use the `.W` or `.N` width specifier to ensure a particular instruction size. `armasm` faults if it cannot generate an instruction with the specified width.

The `.W` specifier is ignored when assembling to A32 code, so you can safely use this specifier in code that might assemble to either A32 or T32 code. However, the `.N` specifier is faulted when assembling to A32 code.

***Related concepts***

*4.10 T32 code size diagnostics* on page 4-85

# Chapter 5
# armasm Command-line Options

Describes the `armasm` command-line syntax and command-line options.

It contains the following sections:

- *5.24 --dwarf3* on page 5-122.
- *5.25 --errors=errorfile* on page 5-123.
- *5.26 --exceptions, --no_exceptions* on page 5-124.
- *5.27 --exceptions_unwind, --no_exceptions_unwind* on page 5-125.
- *5.28 --execstack, --no_execstack* on page 5-126.
- *5.29 --execute_only* on page 5-127.
- *5.30 --fpmode=model* on page 5-128.
- *5.31 --fpu=list* on page 5-129.
- *5.32 --fpu=name* on page 5-130.
- *5.33 -g* on page 5-131.
- *5.34 --help* on page 5-132.
- *5.35 -idir[,dir, ...]* on page 5-133.
- *5.36 --keep* on page 5-134.
- *5.37 --length=n* on page 5-135.
- *5.38 --li* on page 5-136.
- *5.39 --library_type=lib* on page 5-137.
- *5.40 --list=file* on page 5-138.
- *5.41 --list=* on page 5-139.
- *5.42 --littleend* on page 5-140.
- *5.43 -m* on page 5-141.
- *5.44 --maxcache=n* on page 5-142.
- *5.45 --md* on page 5-143.
- *5.46 --no_code_gen* on page 5-144.
- *5.47 --no_esc* on page 5-145.
- *5.48 --no_hide_all* on page 5-146.
- *5.49 --no_regs* on page 5-147.
- *5.50 --no_terse* on page 5-148.
- *5.51 --no_warn* on page 5-149.
- *5.52 -o filename* on page 5-150.
- *5.53 --pd* on page 5-151.
- *5.54 --predefine "directive"* on page 5-152.
- *5.55 --reduce_paths, --no_reduce_paths* on page 5-153.
- *5.56 --regnames* on page 5-154.
- *5.57 --report-if-not-wysiwyg* on page 5-155.
- *5.58 --show_cmdline* on page 5-156.
- *5.59 --thumb* on page 5-157.
- *5.60 --unaligned_access, --no_unaligned_access* on page 5-158.
- *5.61 --unsafe* on page 5-159.
- *5.62 --untyped_local_labels* on page 5-160.
- *5.63 --version_number* on page 5-161.
- *5.64 --via=filename* on page 5-162.
- *5.65 --vsn* on page 5-163.
- *5.66 --width=n* on page 5-164.
- *5.67 --xref* on page 5-165.

## 5.1 --16

Instructs `armasm` to interpret instructions as T32 instructions using the pre-UAL T32 syntax.

This option is equivalent to a `CODE16` directive at the head of the source file. Use the `--thumb` option to specify T32 instructions using the UAL syntax.

**Note**

Not supported for AArch64 state.

***Related reference***

*5.59 --thumb* on page 5-157

*7.11 CODE16 directive* on page 7-213

## 5.2 --32

A synonym for the `--arm` command-line option.

——— **Note** ———

Not supported for AArch64 state.

———————————

***Related reference***

*5.4 --arm* on page 5-100

## 5.3 --apcs=qualifier…qualifier

Controls interworking and position independence when generating code.

### Syntax

`--apcs=`*`qualifier...qualifier`*

Where *`qualifier...qualifier`* denotes a list of qualifiers. There must be:

- At least one qualifier present.
- No spaces or commas separating individual qualifiers in the list.

Each instance of *`qualifier`* must be one of:

**none**

Specifies that the input file does not use AAPCS. AAPCS registers are not set up. Other qualifiers are not permitted if you use `none`.

`/interwork, /nointerwork`

For Armv7-A, Armv7-R, Armv8-A, and Armv8-R, `/interwork` specifies that the code in the input file can interwork between A32 and T32 safely.

The default is `/interwork` for AArch32 targets that support both A32 and T32 instruction sets.

The default is `/nointerwork` for AArch32 targets that only support the T32 instruction set (M-profile targets).

When assembling for AArch64 state, interworking is not available.

`/inter, /nointer`

Are synonyms for `/interwork` and `/nointerwork`.

`/ropi, /noropi`

`/ropi` specifies that the code in the input file is *Read-Only Position-Independent* (ROPI). The default is `/noropi`.

`/pic, /nopic`

Are synonyms for `/ropi` and `/noropi`.

`/rwpi, /norwpi`

`/rwpi` specifies that the code in the input file is *Read-Write Position-Independent* (RWPI). The default is `/norwpi`.

`/pid, /nopid`

Are synonyms for `/rwpi` and `/norwpi`.

`/fpic, /nofpic`

`/fpic` specifies that the code in the input file is read-only independent and references to addresses are suitable for use in a Linux shared object. The default is `/nofpic`.

`/hardfp, /softfp`

Requests hardware or software floating-point linkage. This enables the procedure call standard to be specified separately from the version of the floating-point hardware available through the `--fpu` option. It is still possible to specify the procedure call standard by using the `--fpu` option, but Arm recommends you use `--apcs`. If floating-point support is not permitted (for example, because `--fpu=none` is specified, or because of other means), then `/hardfp` and `/softfp` are ignored. If floating-point support is permitted and the softfp calling convention is used (`--fpu=softvfp` or `--fpu=softvfp+fp-armv8`), then `/hardfp` gives an error.

`/softfp` is not supported for AArch64 state.

### Usage

This option specifies whether you are using the *Procedure Call Standard for the Arm® Architecture* (AAPCS). It can also specify some attributes of code sections.

The AAPCS forms part of the *Base Standard Application Binary Interface for the Arm® Architecture* (BSABI) specification. By writing code that adheres to the AAPCS, you can ensure that separately compiled and assembled modules can work together.

**Note**

AAPCS qualifiers do not affect the code produced by `armasm`. They are an assertion by the programmer that the code in the input file complies with a particular variant of AAPCS. They cause attributes to be set in the object file produced by `armasm`. The linker uses these attributes to check compatibility of files, and to select appropriate library variants.

---

#### Example

```
armasm --cpu=8-A.32 --apcs=/inter/hardfp inputfile.s
```

***Related information***

*Procedure Call Standard for the Arm Architecture*

*Application Binary Interface (ABI) for the Arm Architecture*

## 5.4 --arm

Instructs `armasm` to interpret instructions as A32 instructions. It does not, however, guarantee A32-only code in the object file. This is the default. Using this option is equivalent to specifying the `ARM` or `CODE32` directive at the start of the source file.

**Note**

Not supported for AArch64 state.

***Related reference***

*5.2 --32* on page 5-97

*5.5 --arm_only* on page 5-101

*7.7 ARM or CODE32 directive* on page 7-209

## 5.5 --arm_only

Instructs `armasm` to only generate A32 code. This is similar to `--arm` but also has the property that `armasm` does not permit the generation of any T32 code.

———— **Note** ————

Not supported for AArch64 state.

————————————

***Related reference***

*5.4 --arm* on page 5-100

## 5.6 --bi

A synonym for the `--bigend` command-line option.

***Related reference***

*5.7 --bigend* on page 5-103

*5.42 --littleend* on page 5-140

## 5.7 --bigend

Generates code suitable for an Arm processor using big-endian memory access.

The default is `--littleend`.

***Related reference***

*5.42 --littleend* on page 5-140
*5.6 --bi* on page 5-102

## 5.8 --brief_diagnostics, --no_brief_diagnostics

Enables and disables the output of brief diagnostic messages.

This option instructs the assembler whether to use a shorter form of the diagnostic output. In this form, the original source line is not displayed and the error message text is not wrapped when it is too long to fit on a single line. The default is `--no_brief_diagnostics`.

***Related reference***

*5.17 --diag_error=tag[,tag,...]* on page 5-115

*5.21 --diag_warning=tag[,tag,...]* on page 5-119

## 5.9 --checkreglist

Instructs the `armasm` to check `RLIST`, `LDM`, and `STM` register lists to ensure that all registers are provided in increasing register number order.

When this option is used, `armasm` gives a warning if the registers are not listed in order.

**Note**

In AArch32 state, this option is deprecated. Use `--diag_warning 1206` instead. In AArch64 state, this option is not supported..

***Related reference***

*5.21 --diag_warning=tag[,tag,…]* on page 5-119

## 5.10 --cpreproc

Instructs `armasm` to call `armclang` to preprocess the input file before assembling it.

### Restrictions

You must use `--cpreproc_opts` with this option to correctly configure the `armclang` compiler for pre-processing.

`armasm` only passes the following command-line options to `armclang` by default:

- Basic pre-processor configuration options, such as `-E`.
- User specified include directories, `-I` directives.
- User specified licensing options, such as `--site_license`.
- Anything specified in `--cpreproc_opts`.

***Related concepts***

*4.14 Using the C preprocessor* on page 4-89

***Related reference***

*5.11 --cpreproc_opts=option[,option,…]* on page 5-107

***Related information***

*-x armclang option*

*Command-line options for preprocessing assembly source code*

## 5.11 --cpreproc_opts=option[,option,…]

Enables `armasm` to pass options to `armclang` when using the C preprocessor.

### Syntax

`--cpreproc_opts=`*`option[,option,…]`*

Where `option[,option,…]` is a comma-separated list of C preprocessing options.

At least one option must be specified.

### Restrictions

As a minimum, you must specify the `armclang` options `--target` and either `-mcpu` or `-march` in `--cpreproc_opts`.

To assemble code containing C directives that require the C preprocessor, the input assembly source filename must have an upper-case extension `.S`.

You cannot pass the `armclang` option `-x assembler-with-cpp`, because it gets added to `armclang` after the source file name.

**Note**

Ensure that you specify compatible architectures in the `armclang` options `--target`, `-mcpu` or `-march`, and the `armasm --cpu` option.

### Example

The options to the preprocessor in this example are `--cpreproc_opts=--target=arm-arm-none-eabi,-mcpu=cortex-a9,-D,DEF1,-D,DEF2`.

```
armasm --cpu=cortex-a9 --cpreproc --cpreproc_opts=--target=arm-arm-none-eabi,-mcpu=cortex-
a9,-D,DEF1,-D,DEF2 -I /path/to/includes1 -I /path/to/includes2 input.S
```

***Related concepts***

*4.14 Using the C preprocessor* on page 4-89

***Related reference***

*5.10 --cpreproc* on page 5-106

***Related information***

*Command-line options for preprocessing assembly source code*

*Specifying a target architecture, processor, and instruction set*

*-march armclang option*

*-mcpu armclang option*

*--target armclang option*

*-x armclang option*

## 5.12 --cpu=list

Lists the architecture and processor names that are supported by the `--cpu=name` option.

### Syntax

```
--cpu=list
```

***Related reference***

*5.13 --cpu=name* on page 5-109

## 5.13 --cpu=name

Enables code generation for the selected Arm processor or architecture.

### Syntax

`--cpu=`*`name`*

Where *`name`* is the name of a processor or architecture:

Processor and architecture names are not case-sensitive.

Wildcard characters are not accepted.

The following table shows the supported architectures. For a complete list of the supported architecture and processor names, specify the `--cpu=list` option.

———— **Note** ————

`armasm` does not support architectures later than Armv8.3.

————————————

**Table 5-1 Supported Arm architectures**

| Architecture name | Description |
|---|---|
| `6-M` | Armv6 architecture microcontroller profile. |
| `6S-M` | Armv6 architecture microcontroller profile with OS extensions. |
| `7-A` | Armv7 architecture application profile. |
| `7-A.security` | Armv7-A architecture profile with Security Extensions and includes the `SMC` instruction (formerly `SMI`). |
| `7-R` | Armv7 architecture real-time profile. |
| `7-M` | Armv7 architecture microcontroller profile. |
| `7E-M` | Armv7-M architecture profile with DSP extension. |
| `8-A.32` | Armv8-A architecture profile, AArch32 state. |
| `8-A.32.crypto` | Armv8-A architecture profile, AArch32 state with cryptographic instructions. |
| `8-A.64` | Armv8-A architecture profile, AArch64 state. |
| `8-A.64.crypto` | Armv8-A architecture profile, AArch64 state with cryptographic instructions. |
| `8.1-A.32` | Armv8.1, for Armv8-A architecture profile, AArch32 state. |
| `8.1-A.32.crypto` | Armv8.1, for Armv8-A architecture profile, AArch32 state with cryptographic instructions. |
| `8.1-A.64` | Armv8.1, for Armv8-A architecture profile, AArch64 state. |
| `8.1-A.64.crypto` | Armv8.1, for Armv8-A architecture profile, AArch64 state with cryptographic instructions. |
| `8.2-A.32` | Armv8.2, for Armv8-A architecture profile, AArch32 state. |
| `8.2-A.32.crypto` | Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions. |
| `8.2-A.32.crypto.dotprod` | Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions and the `VSDOT` and `VUDOT` instructions. |
| `8.2-A.32.dotprod` | Armv8.2, for Armv8-A architecture profile, AArch32 state with the `VSDOT` and `VUDOT` instructions. |

**Table 5-1 Supported Arm architectures (continued)**

| Architecture name | Description |
|---|---|
| `8.2-A.64` | Armv8.2, for Armv8-A architecture profile, AArch64 state. |
| `8.2-A.64.crypto` | Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions. |
| `8.2-A.64.crypto.dotprod` | Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions and the `SDOT` and `UDOT` instructions. |
| `8.2-A.64.dotprod` | Armv8.2, for Armv8-A architecture profile, AArch64 state with the `SDOT` and `UDOT` instructions. |
| `8.3-A.32` | Armv8.3, for Armv8-A architecture profile, AArch32 state. |
| `8.3-A.32.crypto` | Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions. |
| `8.3-A.32.crypto.dotprod` | Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions and the `VSDOT` and `VUDOT` instructions. |
| `8.3-A.32.dotprod` | Armv8.3, for Armv8-A architecture profile, AArch32 state with the `VSDOT` and `VUDOT` instructions. |
| `8.3-A.64` | Armv8.3, for Armv8-A architecture profile, AArch64 state. |
| `8.3-A.64.crypto` | Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions. |
| `8.3-A.64.crypto.dotprod` | Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions and the `SDOT` and `UDOT` instructions. |
| `8.3-A.64.dotprod` | Armv8.3, for Armv8-A architecture profile, AArch64 state with the `SDOT` and `UDOT` instructions. |
| `8-R` | Armv8-R architecture profile. |
| `8-M.Base` | Armv8-M baseline architecture profile. Derived from the Armv6-M architecture. |
| `8-M.Main` | Armv8-M mainline architecture profile. Derived from the Armv7-M architecture. |
| `8-M.Main.dsp` | Armv8-M mainline architecture profile with DSP extension. |

**Note**

- The full list of supported architectures and processors depends on your license.

## Default

There is no default option for `--cpu`.

## Usage

The following general points apply to processor and architecture options:

**Processors**

- Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.
- If you specify a processor for the `--cpu` option, the generated code is optimized for that processor. This enables the assembler to use specific coprocessors or instruction scheduling for optimum performance.

**Architectures**

- If you specify an architecture name for the `--cpu` option, the generated code can run on any processor supporting that architecture. For example, `--cpu=7-A` produces code that can be used by the Cortex-A9 processor.

**FPU**

- Some specifications of `--cpu` imply an `--fpu` selection.

  **Note**

  Any explicit FPU, set with `--fpu` on the command line, overrides an implicit FPU.

- If no `--fpu` option is specified and the `--cpu` option does not imply an `--fpu` selection, then `--fpu=softvfp` is used.

**A32/T32**

- Specifying a processor or architecture that supports T32 instructions, such as `--cpu=cortex-a9`, does not make the assembler generate T32 code. It only enables features of the processor to be used, such as long multiply. Use the `--thumb` option to generate T32 code, unless the processor only supports T32 instructions.

  **Note**

  Specifying the target processor or architecture might make the generated object code incompatible with other Arm processors. For example, A32 code generated for architecture Armv8 might not run on a Cortex-A9 processor, if the generated object code includes instructions specific to Armv8. Therefore, you must choose the lowest common denominator processor suited to your purpose.

- If the architecture only supports T32, you do not have to specify `--thumb` on the command line. For example, if building for Cortex-M4 or Armv7-M with `--cpu=7-M`, you do not have to specify `--thumb` on the command line, because Armv7-M only supports T32. Similarly, Armv6-M and other T32-only architectures.

## Restrictions

You cannot specify both a processor and an architecture on the same command-line.

## Example

```
armasm --cpu=Cortex-A17 inputfile.s
```

***Related reference***

*5.3 --apcs=qualifier…qualifier* on page 5-98
*5.12 --cpu=list* on page 5-108
*5.32 --fpu=name* on page 5-130
*5.59 --thumb* on page 5-157
*5.61 --unsafe* on page 5-159

***Related information***

*Arm Architecture Reference Manual*

## 5.14 --debug

Instructs the assembler to generate DWARF debug tables.

`--debug` is a synonym for `-g`. The default is DWARF 3.

**Note**

Local symbols are not preserved with `--debug`. You must specify `--keep` if you want to preserve the local symbols to aid debugging.

***Related reference***

*5.23 --dwarf2* on page 5-121
*5.24 --dwarf3* on page 5-122
*5.36 --keep* on page 5-134
*5.33 -g* on page 5-131

## 5.15 --depend=dependfile

Writes makefile dependency lines to a file.

Source file dependency lists are suitable for use with make utilities.

***Related reference***

*5.45 --md* on page 5-143

*5.16 --depend_format=string* on page 5-114

## 5.16 --depend_format=string

Specifies the format of output dependency files, for compatibility with some UNIX make programs.

### Syntax

`--depend_format=`*`string`*

Where *`string`* is one of:

**`unix`**

generates dependency file entries using UNIX-style path separators.

**`unix_escaped`**

is the same as `unix`, but escapes spaces with `\`.

**`unix_quoted`**

is the same as `unix`, but surrounds path names with double quotes.

***Related reference***

*5.15 --depend=dependfile* on page 5-113

## 5.17 --diag_error=tag[,tag,…]

Sets diagnostic messages that have a specific tag to Error severity.

### Syntax

`--diag_error=tag[,tag,…]`

Where *tag* can be:

- A diagnostic message number to set to error severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `warning`, to treat all warnings as errors.

### Usage

Diagnostic messages output by the assembler can be identified by a tag in the form of {*prefix*}*number*, where the *prefix* is A.

You can specify more than one tag with this option by separating each tag using a comma. You can specify the optional assembler prefix `A` before the tag number. If any prefix other than A is included, the message number is ignored.

The following table shows the meaning of the term severity used in the option descriptions:

**Table 5-2 Severity of diagnostic messages**

| Severity | Description |
|---|---|
| Error | Errors indicate violations in the syntactic or semantic rules of assembly language. Assembly continues, but object code is not generated. |
| Warning | Warnings indicate unusual conditions in your code that might indicate a problem. Assembly continues, and object code is generated unless any problems with an Error severity are detected. |
| Remark | Remarks indicate common, but not recommended, use of assembly language. These diagnostics are not issued by default. Assembly continues, and object code is generated unless any problems with an Error severity are detected. |

***Related reference***

*5.8 --brief_diagnostics, --no_brief_diagnostics* on page 5-104
*5.18 --diag_remark=tag[,tag,…]* on page 5-116
*5.20 --diag_suppress=tag[,tag,…]* on page 5-118
*5.21 --diag_warning=tag[,tag,…]* on page 5-119

## 5.18 --diag_remark=tag[,tag,…]

Sets diagnostic messages that have a specific tag to Remark severity.

### Syntax

`--diag_remark=`*`tag[,tag,…]`*

Where *`tag`* is a comma-separated list of diagnostic message numbers. This is the four-digit number, *`nnnn`*, with the tool letter prefix, but without the letter suffix indicating the severity.

### Usage

Diagnostic messages output by the assembler can be identified by a tag in the form of {*`prefix`*}*`number`*, where the *`prefix`* is A.

You can specify more than one tag with this option by separating each tag using a comma. You can specify the optional assembler prefix `A` before the tag number. If any prefix other than A is included, the message number is ignored.

***Related reference***

*5.8 --brief_diagnostics, --no_brief_diagnostics* on page 5-104
*5.17 --diag_error=tag[,tag,...]* on page 5-115
*5.20 --diag_suppress=tag[,tag,...]* on page 5-118
*5.21 --diag_warning=tag[,tag,...]* on page 5-119

## 5.19 --diag_style={arm|ide|gnu}

Specifies the display style for diagnostic messages.

### Syntax

`--diag_style=`*`string`*

Where *`string`* is one of:

**`arm`**
: Display messages using the legacy Arm compiler style.

**`ide`**
: Include the line number and character count for any line that is in error. These values are displayed in parentheses.

**`gnu`**
: Display messages in the format used by `gcc`.

### Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Choosing the option `--diag_style=ide` implicitly selects the option `--brief_diagnostics`. Explicitly selecting `--no_brief_diagnostics` on the command line overrides the selection of `--brief_diagnostics` implied by `--diag_style=ide`.

Selecting either the option `--diag_style=arm` or the option `--diag_style=gnu` does not imply any selection of `--brief_diagnostics`.

### Default

The default is `--diag_style=arm`.

***Related reference***

*5.8 --brief_diagnostics, --no_brief_diagnostics* on page 5-104

## 5.20 --diag_suppress=tag[,tag,…]

Suppresses diagnostic messages that have a specific tag.

### Syntax

`--diag_suppress=`*`tag[,tag,…]`*

Where *`tag`* can be:

- A diagnostic message number to be suppressed. This is the four-digit number, *`nnnn`*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

Diagnostic messages output by `armasm` can be identified by a tag in the form of {*`prefix`*}*`number`*, where the *`prefix`* is A.

You can specify more than one tag with this option by separating each tag using a comma.

### Example

For example, to suppress the warning messages that have numbers `1293` and `187`, use the following command:

```
armasm --cpu=8-A.64 --diag_suppress=1293,187
```

You can specify the optional assembler prefix `A` before the tag number. For example:

```
armasm --cpu=8-A.64 --diag_suppress=A1293,A187
```

If any prefix other than A is included, the message number is ignored. Diagnostic message tags can be cut and pasted directly into a command line.

***Related reference***

*5.8 --brief_diagnostics, --no_brief_diagnostics* on page 5-104
*5.17 --diag_error=tag[,tag,…]* on page 5-115
*5.18 --diag_remark=tag[,tag,…]* on page 5-116
*5.20 --diag_suppress=tag[,tag,…]* on page 5-118
*5.21 --diag_warning=tag[,tag,…]* on page 5-119

## 5.21 --diag_warning=tag[,tag,…]

Sets diagnostic messages that have a specific tag to Warning severity.

### Syntax

`--diag_warning=tag[,tag,…]`

Where *tag* can be:

- A diagnostic message number to set to warning severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to set all errors that can be downgraded to warnings.

Diagnostic messages output by the assembler can be identified by a tag in the form of {*prefix*}*number*, where the *prefix* is A.

You can specify more than one tag with this option by separating each tag using a comma.

You can specify the optional assembler prefix `A` before the tag number. If any prefix other than A is included, the message number is ignored.

***Related reference***

*5.8 --brief_diagnostics, --no_brief_diagnostics* on page 5-104
*5.17 --diag_error=tag[,tag,...]* on page 5-115
*5.18 --diag_remark=tag[,tag,...]* on page 5-116
*5.20 --diag_suppress=tag[,tag,...]* on page 5-118

## 5.22 --dllexport_all

Controls symbol visibility when building DLLs.

This option gives all exported global symbols `STV_PROTECTED` visibility in ELF rather than `STV_HIDDEN`, unless overridden by source directives.

***Related reference***

*7.27 EXPORT or GLOBAL* on page 7-229

## 5.23 --dwarf2

Uses DWARF 2 debug table format.

**Note**

Not supported for AArch64 state.

---

This option can be used with `--debug`, to instruct `armasm` to generate DWARF 2 debug tables.

***Related reference***

*5.14 --debug* on page 5-112

*5.24 --dwarf3* on page 5-122

## 5.24 --dwarf3

Uses DWARF 3 debug table format.

This option can be used with `--debug`, to instruct the assembler to generate DWARF 3 debug tables. This is the default if `--debug` is specified.

***Related reference***

*5.14 --debug* on page 5-112

*5.23 --dwarf2* on page 5-121

## 5.25 --errors=errorfile

Redirects the output of diagnostic messages from stderr to the specified errors file.

## 5.26 --exceptions, --no_exceptions

Enables or disables exception handling.

**Note**

Not supported for AArch64 state.

---

These options instruct `armasm` to switch on or off exception table generation for all functions defined by `FUNCTION` (or `PROC`) and `ENDFUNC` (or `ENDP`) directives.

`--no_exceptions` causes no tables to be generated. It is the default.

***Related reference***

*5.27 --exceptions_unwind, --no_exceptions_unwind* on page 5-125
*7.39 FRAME UNWIND ON* on page 7-242
*7.40 FRAME UNWIND OFF* on page 7-243
*7.41 FUNCTION or PROC* on page 7-244
*7.24 ENDFUNC or ENDP* on page 7-226

## 5.27 --exceptions_unwind, --no_exceptions_unwind

Enables or disables function unwinding for exception-aware code. This option is only effective if `--exceptions` is enabled.

——— **Note** ———

Not supported for AArch64 state.

———————————

The default is `--exceptions_unwind`.

For finer control, use the `FRAME UNWIND ON` and `FRAME UNWIND OFF` directives.

***Related reference***

*5.26 --exceptions, --no_exceptions* on page 5-124
*7.39 FRAME UNWIND ON* on page 7-242
*7.40 FRAME UNWIND OFF* on page 7-243
*7.41 FUNCTION or PROC* on page 7-244
*7.24 ENDFUNC or ENDP* on page 7-226

## 5.28 --execstack, --no_execstack

Generates a `.note.GNU-stack` section marking the stack as either executable or non-executable.

You can also use the `AREA` directive to generate either an executable or non-executable `.note.GNU-stack` section. The following code generates an executable `.note.GNU-stack` section. Omitting the `CODE` attribute generates a non-executable `.note.GNU-stack` section.

```
    AREA    |.note.GNU-stack|,ALIGN=0,READONLY,NOALLOC,CODE
```

In the absence of `--execstack` and `--no_execstack`, the `.note.GNU-stack` section is not generated unless it is specified by the `AREA` directive.

If both the command-line option and source directive are used and are different, then the stack is marked as executable.

**Table 5-3 Specifying a command-line option and an AREA directive for GNU-stack sections**

| | **`--execstack` command-line option** | **`--no_execstack` command-line option** |
|---|---|---|
| execstack `AREA` directive | execstack | execstack |
| no_execstack `AREA` directive | execstack | no_execstack |

***Related reference***
*7.6 AREA* on page 7-205

## 5.29 --execute_only

Adds the `EXECONLY AREA` attribute to all code sections.

### Usage

The `EXECONLY AREA` attribute causes the linker to treat the section as execute-only.

It is the user's responsibility to ensure that the code in the section is safe to run in execute-only memory. For example:

- The code must not contain literal pools.
- The code must not attempt to load data from the same, or another, execute-only section.

### Restrictions

This option is only supported for:

- Processors that support the Armv8-M mainline or Armv8-M Baseline architecture.
- Processors that support the Armv7-M architecture, such as Cortex-M3, Cortex-M4, and Cortex-M7.
- Processors that support the Armv6-M architecture.

**Note**

Arm has only performed limited testing of execute-only code on Armv6-M targets.

## 5.30 --fpmode=model

Specifies floating-point standard conformance and sets library attributes and floating-point optimizations.

### Syntax

`--fpmode=`*`model`*

Where *`model`* is one of:

**`none`**

Source code is not permitted to use any floating-point type or floating-point instruction. This option overrides any explicit `--fpu=`*`name`* option.

**`ieee_full`**

All facilities, operations, and representations guaranteed by the IEEE standard are available in single and double-precision. Modes of operation can be selected dynamically at runtime.

**`ieee_fixed`**

IEEE standard with round-to-nearest and no inexact exceptions.

**`ieee_no_fenv`**

IEEE standard with round-to-nearest and no exceptions. This mode is compatible with the Java floating-point arithmetic model.

**`std`**

IEEE finite values with denormals flushed to zero, round-to-nearest and no exceptions. It is C and C++ compatible. This is the default option.

Finite values are as predicted by the IEEE standard. It is not guaranteed that NaNs and infinities are produced in all circumstances defined by the IEEE model, or that when they are produced, they have the same sign. Also, it is not guaranteed that the sign of zero is that predicted by the IEEE model.

**`fast`**

Some value altering optimizations, where accuracy is sacrificed to fast execution. This is not IEEE compatible, and is not standard C.

**Note**

This does not cause any changes to the code that you write.

### Example

```
armasm --cpu=8-A.32 --fpmode ieee_full inputfile.s
```

***Related reference***

*5.32 --fpu=name* on page 5-130

***Related information***

*IEEE Standards Association*

## 5.31 --fpu=list

Lists the FPU architecture names that are supported by the `--fpu=name` option.

### Example

```
armasm --fpu=list
```

***Related reference***

*5.30 --fpmode=model* on page 5-128

*5.32 --fpu=name* on page 5-130

## 5.32 --fpu=name

Specifies the target FPU architecture.

### Syntax

`--fpu=name`

Where *name* is the name of the target FPU architecture. Specify `--fpu=list` to list the supported FPU architecture names that you can use with `--fpu=name`.

The default floating-point architecture depends on the target architecture.

**Note**

Software floating-point linkage is not supported for AArch64 state.

### Usage

If you specify this option, it overrides any implicit FPU option that appears on the command line, for example, where you use the `--cpu` option. Floating-point instructions also produce either errors or warnings if assembled for the wrong target FPU.

`armasm` sets a build attribute corresponding to name in the object file. The linker determines compatibility between object files, and selection of libraries, accordingly.

***Related reference***

*5.30 --fpmode=model* on page 5-128

## 5.33 -g

Enables the generation of debug tables.

This option is a synonym for `--debug`.

***Related reference***

*5.14 --debug* on page 5-112

## 5.34 --help

Displays a summary of the main command-line options.

### Default

This is the default if you specify `armasm` without any options or source files.

***Related reference***

*5.63 --version_number* on page 5-161

*5.65 --vsn* on page 5-163

## 5.35 -idir[,dir, …]

Adds directories to the source file include path.

Any directories added using this option have to be fully qualified.

***Related reference***

*7.43 GET or INCLUDE* on page 7-246

## 5.36 --keep

Instructs the assembler to keep named local labels in the symbol table of the object file, for use by the debugger.

***Related reference***

*7.48 KEEP* on page 7-253

## 5.37 --length=n

Sets the listing page length.

Length zero means an unpaged listing. The default is 66 lines.

***Related reference***

*5.40 --list=file* on page 5-138

## 5.38 --li

A synonym for the `--littleend` command-line option.

***Related reference***

*5.42 --littleend* on page 5-140

*5.7 --bigend* on page 5-103

## 5.39 --library_type=lib

Enables the selected library to be used at link time.

### Syntax

`--library_type=`*`lib`*

Where *`lib`* is one of:

**`standardlib`**

Specifies that the full Arm runtime libraries are selected at link time. This is the default.

**`microlib`**

Specifies that the C micro-library (microlib) is selected at link time.

**Note**

- This option can be used with the compiler, assembler, or linker when use of the libraries require more specialized optimizations.
- This option can be overridden at link time by providing it to the linker.
- microlib is not supported for AArch64 state.

***Related information***

*Building an application with microlib*

## 5.40 --list=file

Instructs the assembler to output a detailed listing of the assembly language produced by the assembler to a file.

If - is given as `file`, the listing is sent to `stdout`.

Use the following command-line options to control the behavior of `--list`:

- `--no_terse`.
- `--width`.
- `--length`.
- `--xref`.

***Related reference***

*5.50 --no_terse* on page 5-148
*5.66 --width=n* on page 5-164
*5.37 --length=n* on page 5-135
*5.67 --xref* on page 5-165
*7.55 OPT* on page 7-262

## 5.41 --list=

Instructs the assembler to send the detailed assembly language listing to *`inputfile`*`.lst`.

**Note**

You can use `--list` without the equals sign and filename to send the output to *`inputfile`*`.lst`. However, this syntax is deprecated and the assembler issues a warning. This syntax is to be removed in a later release. Use `--list=` instead.

***Related reference***

*5.40 --list=file* on page 5-138

## 5.42 --littleend

Generates code suitable for an Arm processor using little-endian memory access.

***Related reference***

*5.7 --bigend* on page 5-103

*5.38 --li* on page 5-136

## 5.43 -m

Instructs the assembler to write source file dependency lists to `stdout`.

***Related reference***

*5.45 --md* on page 5-143

## 5.44 --maxcache=n

Sets the maximum source cache size in bytes.

The default is 8MB. `armasm` gives a warning if the size is less than 8MB.

## 5.45 --md

Creates makefile dependency lists.

This option instructs the assembler to write source file dependency lists to `inputfile.d`.

***Related reference***

*5.43 -m* on page 5-141

## 5.46 --no_code_gen

Instructs the assembler to exit after pass 1, generating no object file. This option is useful if you only want to check the syntax of the source code or directives.

## 5.47 --no_esc

Instructs the assembler to ignore C-style escaped special characters, such as \n and \t.

## 5.48 --no_hide_all

Gives all exported and imported global symbols `STV_DEFAULT` visibility in ELF rather than `STV_HIDDEN`, unless overridden using source directives.

You can use the following directives to specify an attribute that overrides the implicit symbol visibility:

- `EXPORT`.
- `EXTERN`.
- `GLOBAL`.
- `IMPORT`.

***Related reference***

*7.27 EXPORT or GLOBAL* on page 7-229

*7.45 IMPORT and EXTERN* on page 7-249

## 5.49 --no_regs

Instructs `armasm` not to predefine register names.

**Note**

This option is deprecated. In AArch32 state, use `--regnames=none` instead.

***Related reference***

*5.56 --regnames* on page 5-154

## 5.50 --no_terse

Instructs the assembler to show in the list file the lines of assembly code that it has skipped because of conditional assembly.

If you do not specify this option, the assembler does not output the skipped assembly code to the list file.

This option turns off the terse flag. By default the terse flag is on.

***Related reference***

*5.40 --list=file* on page 5-138

## 5.51 --no_warn

Turns off warning messages.

***Related reference***

*5.21 --diag_warning=tag[,tag,…]* on page 5-119

## 5.52 -o filename

Specifies the name of the output file.

If this option is not used, the assembler creates an object filename in the form `inputfilename.o`. This option is case-sensitive.

## 5.53 --pd

A synonym for the `--predefine` command-line option.

***Related reference***

*5.54 --predefine "directive"* on page 5-152

## 5.54 --predefine "directive"

Instructs `armasm` to pre-execute one of the `SETA`, `SETL`, or `SETS` directives.

You must enclose *`directive`* in quotes, for example:

```
armasm --cpu=8-A.64 --predefine "VariableName SETA 20" inputfile.s
```

`armasm` also executes a corresponding `GBLL`, `GBLS`, or `GBLA` directive to define the variable before setting its value.

The variable name is case-sensitive. The variables defined using the command line are global to `armasm` source files specified on the command line.

### Considerations when using --predefine

Be aware of the following:

- The command-line interface of your system might require you to enter special character combinations, such as `\"`, to include strings in *`directive`*. Alternatively, you can use `--via` *`file`* to include a `--predefine` argument. The command-line interface does not alter arguments from `--via` files.
- `--predefine` is not equivalent to the compiler option `-D`*`name`*. `--predefine` defines a global variable whereas `-D`*`name`* defines a macro that the C preprocessor expands.

  Although you can use predefined global variables in combination with assembly control directives, for example `IF` and `ELSE` to control conditional assembly, they are not intended to provide the same functionality as the C preprocessor in `armasm`. If you require this functionality, Arm recommends you use the compiler to pre-process your assembly code.

***Related reference***

*5.53 --pd* on page 5-151
*7.42 GBLA, GBLL, and GBLS* on page 7-245
*7.44 IF, ELSE, ENDIF, and ELIF* on page 7-247
*7.63 SETA, SETL, and SETS* on page 7-272

## 5.55 --reduce_paths, --no_reduce_paths

Enables or disables the elimination of redundant path name information in file paths.

Windows systems impose a 260 character limit on file paths. Where relative pathnames exist whose absolute names expand to longer than 260 characters, you can use the `--reduce_paths` option to reduce absolute pathname length by matching up directories with corresponding instances of `..` and eliminating the directory/`..` sequences in pairs.

`--no_reduce_paths` is the default.

———— **Note** ————

Arm recommends that you avoid using long and deeply nested file paths, in preference to minimizing path lengths using the `--reduce_paths` option.

————————————

———— **Note** ————

This option is valid for 32-bit Windows systems only.

————————————

## 5.56 --regnames

Controls the predefinition of register names.

**Note**

Not supported for AArch64 state.

### Syntax

`--regnames=`*`option`*

Where *`option`* is one of the following:

**none**
Instructs `armasm` not to predefine register names.

**callstd**
Defines additional register names based on the AAPCS variant that you are using, as specified by the `--apcs` option.

**all**
Defines all AAPCS registers regardless of the value of `--apcs`.

***Related reference***

*5.49 --no_regs* on page 5-147
*5.56 --regnames* on page 5-154
*5.3 --apcs=qualifier…qualifier* on page 5-98

## 5.57 --report-if-not-wysiwyg

Instructs `armasm` to report when it outputs an encoding that was not directly requested in the source code.

This can happen when `armasm`:

- Uses a pseudo-instruction that is not available in other assemblers, for example `MOV32`.
- Outputs an encoding that does not directly match the instruction mnemonic, for example if the assembler outputs the `MVN` encoding when assembling the `MOV` instruction.
- Inserts additional instructions where necessary for instruction syntax semantics, for example `armasm` can insert a missing `IT` instruction before a conditional T32 instruction.

**Note**

Not supported for AArch64 state.

## 5.58 --show_cmdline

Outputs the command line used by the assembler.

### Usage

Shows the command line after processing by the assembler, and can be useful to check:

- The command line a build system is using.
- How the assembler is interpreting the supplied command line, for example, the ordering of command-line options.

The commands are shown normalized, and the contents of any via files are expanded.

The output is sent to the standard error stream (`stderr`).

***Related reference***

*5.64 --via=filename* on page 5-162

## 5.59 --thumb

Instructs `armasm` to interpret instructions as T32 instructions, using UAL syntax. This is equivalent to a `THUMB` directive at the start of the source file.

———— **Note** ————

Not supported for AArch64 state.

————————————

***Related reference***

*5.4 --arm* on page 5-100

*7.65 THUMB directive* on page 7-275

## 5.60 --unaligned_access, --no_unaligned_access

Enables or disables unaligned accesses to data on Arm-based processors.

These options instruct the assembler to set an attribute in the object file to enable or disable the use of unaligned accesses.

## 5.61 --unsafe

Enables instructions for other architectures to be assembled without error.

———— **Note** ————

Not supported for AArch64 state.

————————————

It downgrades error messages to corresponding warning messages. It also suppresses warnings about operator precedence.

***Related concepts***

*6.20 Binary operators* on page 6-187

***Related reference***

*5.17 --diag_error=tag[,tag,...]* on page 5-115

*5.21 --diag_warning=tag[,tag,...]* on page 5-119

## 5.62 --untyped_local_labels

Causes `armasm` not to set the T32 bit for the address of a numeric local label referenced in an `LDR` pseudo-instruction.

**Note**

Not supported for AArch64 state.

When this option is not used, if you reference a numeric local label in an `LDR` pseudo-instruction, and the label is in T32 code, then `armasm` sets the T32 bit (bit 0) of the address. You can then use the address as the target for a `BX` or `BLX` instruction.

If you require the actual address of the numeric local label, without the T32 bit set, then use this option.

**Note**

When using this option, if you use the address in a branch (register) instruction, `armasm` treats it as an A32 code address, causing the branch to arrive in A32 state, meaning it would interpret this code as A32 instructions.

### Example

```
THUMB
    ...
1
    ...
    LDR r0,=%B1 ; r0 contains the address of numeric local label "1",
                ; T32 bit is not set if --untyped_local_labels was used
    ...
```

***Related concepts***

*6.10 Numeric local labels* on page 6-177

## 5.63 --version_number

Displays the version of `armasm` you are using.

### Usage

The assembler displays the version number in the format `Mmmuuxx`, where:

- *`M`* is the major version number, `6`.
- *`mm`* is the minor version number.
- *`uu`* is the update number.
- *`xx`* is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.

## 5.64 --via=filename

Reads an additional list of input filenames and assembler options from `filename`.

### Syntax

`--via=filename`

Where `filename` is the name of a via file containing options to be included on the command line.

### Usage

You can enter multiple `--via` options on the assembler command line. The `--via` options can also be included within a via file.

***Related concepts***

*9.1 Overview of via files* on page 9-291

***Related reference***

*9.2 Via file syntax rules* on page 9-292

## 5.65 --vsn

Displays the version information and the license details.

**Note**

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm Compiler you are using. If you need to access the version in other tools or scripts, for example in build scripts, use the output from `--version_number`.

### Example

```
> armasm --vsn
Product: ARM Compiler N.n
Component: ARM Compiler N.n
Tool: armasm [tool_id]
license_type
Software supplied by: ARM Limited
```

## 5.66 --width=n

Sets the listing page width.

The default is 79 characters.

***Related reference***

*5.40 --list=file* on page 5-138

## 5.67 --xref

Instructs the assembler to list cross-referencing information on symbols, including where they were defined and where they were used, both inside and outside macros.

The default is off.

***Related reference***

*5.40 --list=file* on page 5-138

# Chapter 6
# Symbols, Literals, Expressions, and Operators

Describes how you can use symbols to represent variables, addresses, and constants in code, and how you can combine these with operators to create numeric or string expressions.

It contains the following sections:

- *6.23 Shift operators* on page 6-190.
- *6.24 Addition, subtraction, and logical operators* on page 6-191.
- *6.25 Relational operators* on page 6-192.
- *6.26 Boolean operators* on page 6-193.
- *6.27 Operator precedence* on page 6-194.
- *6.28 Difference between operator precedence in assembly language and C* on page 6-195.

## 6.1 Symbol naming rules

You must follow some rules when naming symbols in assembly language source code.

The following rules apply:

- Symbol names must be unique within their scope.
- You can use uppercase letters, lowercase letters, numeric characters, or the underscore character in symbol names. Symbol names are case-sensitive, and all characters in the symbol name are significant.
- Do not use numeric characters for the first character of symbol names, except in numeric local labels.
- Symbols must not use the same name as built-in variable names or predefined symbol names.
- If you use the same name as an instruction mnemonic or directive, use double bars to delimit the symbol name. For example:

  ```
  ||ASSERT||
  ```

  The bars are not part of the symbol.
- You must not use the symbols `|$a|`, `|$t|`, or `|$d|` as program labels. These are mapping symbols that mark the beginning of A32, T32, and A64 code, and data within the object file. You must not use `|$x|` in A64 code.
- Symbols beginning with the characters $v are mapping symbols that relate to floating-point code. Arm recommends you avoid using symbols beginning with $v in your source code.

If you have to use a wider range of characters in symbols, for example, when working with compilers, use single bars to delimit the symbol name. For example:

```
|.text|
```

The bars are not part of the symbol. You cannot use bars, semicolons, or newlines within the bars.

***Related concepts***

*6.10 Numeric local labels* on page 6-177

***Related reference***

*4.4 Built-in variables and constants* on page 4-76

## 6.2 Variables

You can declare numeric, logical, or string variables using assembler directives.

The value of a variable can be changed as assembly proceeds. Variables are local to the assembler. This means that in the generated code or data, every instance of the variable has a fixed value.

The type of a variable cannot be changed. Variables are one of the following types:
- Numeric.
- Logical.
- String.

The range of possible values of a numeric variable is the same as the range of possible values of a numeric constant or numeric expression.

The possible values of a logical variable are `{TRUE}` or `{FALSE}`.

The range of possible values of a string variable is the same as the range of values of a string expression.

Use the `GBLA`, `GBLL`, `GBLS`, `LCLA`, `LCLL`, and `LCLS` directives to declare symbols representing variables, and assign values to them using the `SETA`, `SETL`, and `SETS` directives.

### Example

```
a    SETA 100
L1   MOV R1, #(a*5) ; In the object file, this is MOV R1, #500
a    SETA 200       ; Value of 'a' is 200 only after this point.
                    ; The previous instruction is always MOV R1, #500
     …
     BNE L1         ; When the processor branches to L1, it executes
                    ; MOV R1, #500
```

***Related concepts***

*6.14 Numeric expressions* on page 6-181
*6.12 String expressions* on page 6-179
*6.3 Numeric constants* on page 6-170
*6.17 Logical expressions* on page 6-184

***Related reference***

*7.42 GBLA, GBLL, and GBLS* on page 7-245
*7.49 LCLA, LCLL, and LCLS* on page 7-254
*7.63 SETA, SETL, and SETS* on page 7-272

## 6.3 Numeric constants

You can define 32-bit numeric constants using the `EQU` assembler directive.

Numeric constants are 32-bit integers in A32 and T32 code. You can set them using unsigned numbers in the range 0 to $2^{32}$-1, or signed numbers in the range -$2^{31}$ to $2^{31}$ -1. However, the assembler makes no distinction between -*n* and $2^{32}$-*n*.

In A64 code, numeric constants are 64-bit integers. You can set them using unsigned numbers in the range 0 to $2^{64}$-1, or signed numbers in the range -$2^{63}$ to $2^{63}$-1. However, the assembler makes no distinction between -*n* and $2^{64}$-*n*.

Relational operators such as >= use the unsigned interpretation. This means that 0 > -1 is `{FALSE}`.

Use the `EQU` directive to define constants. You cannot change the value of a numeric constant after you define it. You can construct expressions by combining numeric constants and binary operators.

***Related concepts***

*6.14 Numeric expressions* on page 6-181

***Related reference***

*6.15 Syntax of numeric literals* on page 6-182

*7.26 EQU* on page 7-228

## 6.4 Assembly time substitution of variables

You can assign a string variable to all or part of a line of assembly language code. A string variable can contain numeric and logical variables.

Use the variable with a `$` prefix in the places where the value is to be substituted for the variable. The dollar character instructs `armasm` to substitute the string into the source code line before checking the syntax of the line. `armasm` faults if the substituted line is larger than the source line limit.

Numeric and logical variables can also be substituted. The current value of the variable is converted to a hexadecimal string (or `T` or `F` for logical variables) before substitution.

Use a dot to mark the end of the variable name if the following character would be permissible in a symbol name. You must set the contents of the variable before you can use it.

If you require a `$` that you do not want to be substituted, use `$$`. This is converted to a single `$`.

You can include a variable with a `$` prefix in a string. Substitution occurs in the same way as anywhere else.

Substitution does not occur within vertical bars, except that vertical bars within double quotes do not affect substitution.

### Example

```
    ; straightforward substitution
            GBLS    add4ff
            ;
add4ff      SETS    "ADD  r4,r4,#0xFF"    ; set up add4ff
            $add4ff.00                    ; invoke add4ff
            ; this produces
            ADD  r4,r4,#0xFF00
    ; elaborate substitution
            GBLS    s1
            GBLS    s2
            GBLS    fixup
            GBLA    count
            ;
count       SETA    14
s1          SETS    "a$$b$count" ; s1 now has value a$b0000000E
s2          SETS    "abc"
fixup       SETS    "|xy$s2.z|"  ; fixup now has value |xyabcz|
|C$$code|   MOV     r4,#16       ; but the label here is C$$code
```

***Related reference***

*2.1 Syntax of source lines in assembly language* on page 2-28

*6.1 Symbol naming rules* on page 6-168

## 6.5 Register-relative and PC-relative expressions

The assembler supports PC-relative and register-relative expressions.

A register-relative expression evaluates to a named register combined with a numeric expression.

You write a PC-relative expression in source code as a label or the PC, optionally combined with a numeric expression. Some instructions can also accept PC-relative expressions in the form `[PC, #number]`.

If you specify a label, the assembler calculates the offset from the PC value of the current instruction to the address of the label. The assembler encodes the offset in the instruction. If the offset is too large, the assembler produces an error. The offset is either added to or subtracted from the PC value to form the required address.

Arm recommends you write PC-relative expressions using labels rather than the PC because the value of the PC depends on the instruction set.

———— **Note** ————

- In A32 code, the value of the PC is the address of the current instruction plus 8 bytes.
- In T32 code:
  — For `B`, `BL`, `CBNZ`, and `CBZ` instructions, the value of the PC is the address of the current instruction plus 4 bytes.
  — For all other instructions that use labels, the value of the PC is the address of the current instruction plus 4 bytes, with bit[1] of the result cleared to 0 to make it word-aligned.
- In A64 code, the value of the PC is the address of the current instruction.

————————————

### Example

```
        LDR     r4,=data+4*n    ; n is an assembly-time variable
        ; code
        MOV     pc,lr
data    DCD     value_0
        ; n-1 DCD directives
        DCD     value_n         ; data+4*n points here
        ; more DCD directives
```

***Related concepts***

*6.6 Labels* on page 6-173

***Related reference***

*7.52 MAP* on page 7-259

## 6.6 Labels

A label is a symbol that represents the memory address of an instruction or data.

The address can be PC-relative, register-relative, or absolute. Labels are local to the source file unless you make them global using the `EXPORT` directive.

The address given by a label is calculated during assembly. `armasm` calculates the address of a label relative to the origin of the section where the label is defined. A reference to a label within the same section can use the PC plus or minus an offset. This is called *PC-relative addressing*.

Addresses of labels in other sections are calculated at link time, when the linker has allocated specific locations in memory for each section.

***Related concepts***

*6.7 Labels for PC-relative addresses* on page 6-174

*6.8 Labels for register-relative addresses* on page 6-175

*6.9 Labels for absolute addresses* on page 6-176

***Related reference***

*2.1 Syntax of source lines in assembly language* on page 2-28

*7.27 EXPORT or GLOBAL* on page 7-229

## 6.7 Labels for PC-relative addresses

A label can represent the PC value plus or minus the offset from the PC to the label. Use these labels as targets for branch instructions, or to access small items of data embedded in code sections.

You can define PC-relative labels using a label on an instruction or on one of the data definition directives.

You can also use the section name of an `AREA` directive as a label for PC-relative addresses. In this case the label points to the first byte of the specified `AREA`. Arm does not recommend using `AREA` names as branch targets because when branching from A32 to T32 state or T32 to A32 state in this way, the processor does not change the state properly.

***Related reference***

*7.6 AREA* on page 7-205
*7.15 DCB* on page 7-217
*7.16 DCD and DCDU* on page 7-218
*7.18 DCFD and DCFDU* on page 7-220
*7.19 DCFS and DCFSU* on page 7-221
*7.20 DCI* on page 7-222
*7.21 DCQ and DCQU* on page 7-223
*7.22 DCW and DCWU* on page 7-224

## 6.8 Labels for register-relative addresses

A label can represent a named register plus a numeric value. You define these labels in a storage map. They are most commonly used to access data in data sections.

You can use the `EQU` directive to define additional register-relative labels, based on labels defined in storage maps.

**Note**

Register-relative addresses are not supported in A64 code.

### Example of storage map definitions

```
        MAP     0,r9
        MAP     0xff,r9
```

***Related reference***

*7.17 DCDO* on page 7-219

*7.26 EQU* on page 7-228

*7.52 MAP* on page 7-259

*7.64 SPACE or FILL* on page 7-274

## 6.9 Labels for absolute addresses

A label can represent the absolute address of code or data.

These labels are numeric constants. In A32 and T32 code they are integers in the range 0 to $2^{32}$-1. In A64 code, they are integers in the range 0 to $2^{64}$-1. They address the memory directly. You can use labels to represent absolute addresses using the `EQU` directive. To ensure that the labels are used correctly when referenced in code, you can specify the absolute address as:

- A32 code with the `ARM` directive.
- T32 code with the `THUMB` directive.
- Data.

### Example of defining labels for absolute address

```
abc EQU 2               ; assigns the value 2 to the symbol abc
xyz EQU label+8         ; assigns the address (label+8) to the symbol xyz
fiq EQU 0x1C, ARM       ; assigns the absolute address 0x1C to the symbol fiq
                        ; and marks it as A32 code
```

***Related concepts***

*6.6 Labels* on page 6-173

*6.7 Labels for PC-relative addresses* on page 6-174

*6.8 Labels for register-relative addresses* on page 6-175

***Related reference***

*7.26 EQU* on page 7-228

## 6.10 Numeric local labels

Numeric local labels are a type of label that you refer to by number rather than by name. They are used in a similar way to PC-relative labels, but their scope is more limited.

A numeric local label is a number in the range 0-99, optionally followed by a name. Unlike other labels, a numeric local label can be defined many times and the same number can be used for more than one numeric local label in an area.

Numeric local labels do not appear in the object file. This means that, for example, a debugger cannot set a breakpoint directly on a numeric local label, like it can for named local labels kept using the `KEEP` directive.

A numeric local label can be used in place of *`symbol`* in source lines in an assembly language module:

- On its own, that is, where there is no instruction or directive.
- On a line that contains an instruction.
- On a line that contains a code- or data-generating directive.

A numeric local label is generally used where you might use a PC-relative label.

Numeric local labels are typically used for loops and conditional code within a routine, or for small subroutines that are only used locally. They are particularly useful when you are generating labels in macros.

The scope of numeric local labels is limited by the `AREA` directive. Use the `ROUT` directive to limit the scope of numeric local labels more tightly. A reference to a numeric local label refers to a matching label within the same scope. If there is no matching label within the scope in either direction, `armasm` generates an error message and the assembly fails.

You can use the same number for more than one numeric local label even within the same scope. By default, `armasm` links a numeric local label reference to:
- The most recent numeric local label with the same number, if there is one within the scope.
- The next following numeric local label with the same number, if there is not a preceding one within the scope.

Use the optional parameters to modify this search pattern if required.

***Related concepts***

*6.6 Labels* on page 6-173

***Related reference***

*2.1 Syntax of source lines in assembly language* on page 2-28

*6.11 Syntax of numeric local labels* on page 6-178

*7.51 MACRO and MEND* on page 7-256

*7.48 KEEP* on page 7-253

*7.62 ROUT* on page 7-271

## 6.11 Syntax of numeric local labels

When referring to numeric local labels you can specify how `armasm` searches for the label.

### Syntax

```
n[routname] ; a numeric local label
```

```
%[F|B][A|T]n[routname] ; a reference to a numeric local label
```

where:

*n*

is the number of the numeric local label in the range 0-99.

*routname*

is the name of the current scope.

**%**

introduces the reference.

**F**

instructs `armasm` to search forwards only.

**B**

instructs `armasm` to search backwards only.

**A**

instructs `armasm` to search all macro levels.

**T**

instructs `armasm` to look at this macro level only.

### Usage

If neither `F` nor `B` is specified, `armasm` searches backwards first, then forwards.

If neither `A` nor `T` is specified, `armasm` searches all macros from the current level to the top level, but does not search lower level macros.

If *routname* is specified in either a label or a reference to a label, `armasm` checks it against the name of the nearest preceding `ROUT` directive. If it does not match, `armasm` generates an error message and the assembly fails.

***Related concepts***

*6.10 Numeric local labels* on page 6-177

***Related reference***

*7.62 ROUT* on page 7-271

## 6.12 String expressions

String expressions consist of combinations of string literals, string variables, string manipulation operators, and parentheses.

Characters that cannot be placed in string literals can be placed in string expressions using the `:CHR:` unary operator. Any ASCII character from 0 to 255 is permitted.

The value of a string expression cannot exceed 5120 characters in length. It can be of zero length.

### Example

```
improb  SETS    "literal":CC:(strvar2:LEFT:4)
            ; sets the variable improb to the value "literal"
            ; with the left-most four characters of the
            ; contents of string variable strvar2 appended
```

***Related concepts***

*6.13 String literals* on page 6-180

*6.19 Unary operators* on page 6-186

*6.2 Variables* on page 6-169

***Related reference***

*6.22 String manipulation operators* on page 6-189

*7.63 SETA, SETL, and SETS* on page 7-272

## 6.13 String literals

String literals consist of a series of characters or spaces contained between double quote characters.

The length of a string literal is restricted by the length of the input line.

To include a double quote character or a dollar character within the string literal, include the character twice as a pair. For example, you must use `$$` if you require a single `$` in the string.

C string escape sequences are also enabled and can be used within the string, unless `--no_esc` is specified.

### Examples

```
abc     SETS    "this string contains only one "" double quote"
def     SETS    "this string contains only one $$ dollar symbol"
```

***Related reference***

*2.1 Syntax of source lines in assembly language* on page 2-28

*5.47 --no_esc* on page 5-145

## 6.14 Numeric expressions

Numeric expressions consist of combinations of numeric constants, numeric variables, ordinary numeric literals, binary operators, and parentheses.

Numeric expressions can contain register-relative or program-relative expressions if the overall expression evaluates to a value that does not include a register or the PC.

Numeric expressions evaluate to 32-bit integers in A32 and T32 code. You can interpret them as unsigned numbers in the range 0 to $2^{32}$-1, or signed numbers in the range -$2^{31}$ to $2^{31}$-1. However, `armasm` makes no distinction between -*n* and $2^{32}$-*n*. Relational operators such as >= use the unsigned interpretation. This means that 0 > -1 is `{FALSE}`.

In A64 code, numeric expressions evaluate to 64-bit integers. You can interpret them as unsigned numbers in the range 0 to $2^{64}$-1, or signed numbers in the range -$2^{63}$ to $2^{63}$-1. However, `armasm` makes no distinction between -*n* and $2^{64}$-*n*.

**Note**

`armasm` does not support 64-bit arithmetic variables. See *7.63 SETA, SETL, and SETS* on page 7-272 (Restrictions) for a workaround.

Arm recommends that you only use `armasm` for legacy Arm syntax assembly code, and that you use the `armclang` assembler and GNU syntax for all new assembly files.

### Example

```
a    SETA    256*256            ; 256*256 is a numeric expression
     MOV     r1,#(a*22)         ; (a*22) is a numeric expression
```

***Related concepts***

*6.20 Binary operators* on page 6-187

*6.2 Variables* on page 6-169

*6.3 Numeric constants* on page 6-170

***Related reference***

*6.15 Syntax of numeric literals* on page 6-182

*7.63 SETA, SETL, and SETS* on page 7-272

## 6.15 Syntax of numeric literals

Numeric literals consist of a sequence of characters, or a single character in quotes, evaluating to an integer.

They can take any of the following forms:

- *`decimal-digits`*.
- `0x`*`hexadecimal-digits`*.
- `&`*`hexadecimal-digits`*.
- *`n_base-n-digits`*.
- `'`*`character`*`'`.

where:

*`decimal-digits`*
Is a sequence of characters using only the digits 0 to 9.

*`hexadecimal-digits`*
Is a sequence of characters using only the digits 0 to 9 and the letters A to F or a to f.

*`n_`*
Is a single digit between 2 and 9 inclusive, followed by an underscore character.

*`base-n-digits`*
Is a sequence of characters using only the digits 0 to (*n*-1)

*`character`*
Is any single character except a single quote. Use the standard C escape character (\') if you require a single quote. The character must be enclosed within opening and closing single quotes. In this case, the value of the numeric literal is the numeric code of the character.

You must not use any other characters. The sequence of characters must evaluate to an integer.

In A32/T32 code, the range is 0 to $2^{32}$-1, except in `DCQ`, `DCQU`, `DCD`, and `DCDU` directives.

In A64 code, the range is 0 to $2^{64}$-1, except in `DCD` and `DCDU` directives.

**Note**

- In the `DCQ` and `DCQU`, the integer range is 0 to $2^{64}$-1
- In the `DCO` and `DCOU` directives, the integer range is 0 to $2^{128}$-1

### Examples

```
a       SETA    34906
addr    DCD     0xA10E
        LDR     r4,=&1000000F
        DCD     2_11001010
c3      SETA    8_74007
        DCQ     0x0123456789abcdef
        LDR     r1,='A'      ; pseudo-instruction loading 65 into r1
        ADD     r3,r2,#'\''  ; add 39 to contents of r2, result to r3
```

***Related concepts***

*6.3 Numeric constants* on page 6-170

## 6.16 Syntax of floating-point literals

Floating-point literals consist of a sequence of characters evaluating to a floating-point number.

They can take any of the following forms:

- `{-}`*`digits`*`E{-}`*`digits`*
- `{-}{`*`digits`*`}.digits`
- `{-}{`*`digits`*`}.digitsE{-}`*`digits`*
- `0x`*`hexdigits`*
- `&`*`hexdigits`*
- `0f_`*`hexdigits`*
- `0d_`*`hexdigits`*

where:

*`digits`*
: Are sequences of characters using only the digits 0 to 9. You can write `E` in uppercase or lowercase. These forms correspond to normal floating-point notation.

*`hexdigits`*
: Are sequences of characters using only the digits 0 to 9 and the letters A to F or a to f. These forms correspond to the internal representation of the numbers in the computer. Use these forms to enter infinities and NaNs, or if you want to be sure of the exact bit patterns you are using.

The `0x` and `&` forms allow the floating-point bit pattern to be specified by any number of hex digits.

The `0f_` form requires the floating-point bit pattern to be specified by exactly 8 hex digits.

The `0d_` form requires the floating-point bit pattern to be specified by exactly 16 hex digits.

The range for half-precision floating-point values is:

- Maximum 65504 (IEEE format) or 131008 (alternative format).
- Minimum 0.00012201070785522461.

The range for single-precision floating-point values is:

- Maximum 3.40282347e+38.
- Minimum 1.17549435e-38.

The range for double-precision floating-point values is:

- Maximum 1.79769313486231571e+308.
- Minimum 2.22507385850720138e-308.

Floating-point numbers are only available if your system has floating-point, Advanced SIMD with floating-point.

### Examples

```
    DCFD    1E308,-4E-100
    DCFS    1.0
    DCFS    0.02
    DCFD    3.725e15
    DCFS    0x7FC00000              ; Quiet NaN
    DCFD    &FFF0000000000000       ; Minus infinity
```

***Related concepts***

*6.3 Numeric constants* on page 6-170

***Related reference***

*6.15 Syntax of numeric literals* on page 6-182

## 6.17 Logical expressions

Logical expressions consist of combinations of logical literals (`{TRUE}` or `{FALSE}`), logical variables, Boolean operators, relations, and parentheses.

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators.

***Related reference***

*6.26 Boolean operators* on page 6-193

*6.25 Relational operators* on page 6-192

## 6.18 Logical literals

Logical or Boolean literals can have one of two values, `{TRUE}` or `{FALSE}`.

***Related concepts***

*6.13 String literals* on page 6-180

***Related reference***

*6.15 Syntax of numeric literals* on page 6-182

## 6.19 Unary operators

Unary operators return a string, numeric, or logical value. They have higher precedence than other operators and are evaluated first.

A unary operator precedes its operand. Adjacent operators are evaluated from right to left.

The following table lists the unary operators that return strings:

**Table 6-1 Unary operators that return strings**

| Operator | Usage | Description |
|---|---|---|
| `:CHR:` | `:CHR:A` | Returns the character with ASCII code A. |
| `:LOWERCASE:` | `:LOWERCASE:string` | Returns the given string, with all uppercase characters converted to lowercase. |
| `:REVERSE_CC:` | `:REVERSE_CC:cond_code` | Returns the inverse of the condition code in `cond_code`, or an error if `cond_code` does not contain a valid condition code. |
| `:STR:` | `:STR:A` | In A32 and T32 code, returns an 8-digit hexadecimal string corresponding to a numeric expression, or the string `"T"` or `"F"` if used on a logical expression. In A64 code, returns a 16-digit hexadecimal string. |
| `:UPPERCASE:` | `:UPPERCASE:string` | Returns the given string, with all lowercase characters converted to uppercase. |

The following table lists the unary operators that return numeric values:

**Table 6-2 Unary operators that return numeric or logical values**

| Operator | Usage | Description |
|---|---|---|
| `?` | `?A` | Number of bytes of code generated by line defining symbol A. |
| `+` and `-` | `+A` `-A` | Unary plus. Unary minus. + and – can act on numeric and PC-relative expressions. |
| `:BASE:` | `:BASE:A` | If A is a PC-relative or register-relative expression, `:BASE:` returns the number of its register component. `:BASE:` is most useful in macros. |
| `:CC_ENCODING:` | `:CC_ENCODING:cond_code` | Returns the numeric value of the condition code in `cond_code`, or an error if `cond_code` does not contain a valid condition code. |
| `:DEF:` | `:DEF:A` | {`TRUE`} if A is defined, otherwise {`FALSE`}. |
| `:INDEX:` | `:INDEX:A` | If A is a register-relative expression, `:INDEX:` returns the offset from that base register. `:INDEX:` is most useful in macros. |
| `:LEN:` | `:LEN:A` | Length of string A. |
| `:LNOT:` | `:LNOT:A` | Logical complement of A. |
| `:NOT:` | `:NOT:A` | Bitwise complement of A (~ is an alias, for example `~A`). |
| `:RCONST:` | `:RCONST:Rn` | Number of register. In A32/T32 code, 0-15 corresponds to R0-R15. In A64 code, 0-30 corresponds to W0-W30 or X0-X30. |

***Related concepts***

*6.20 Binary operators* on page 6-187

## 6.20 Binary operators

You write binary operators between the pair of sub-expressions they operate on. They have lower precedence than unary operators.

——— **Note** ———

The order of precedence is not the same as in C.

———————————

***Related concepts***

*6.28 Difference between operator precedence in assembly language and C* on page 6-195

***Related reference***

*6.21 Multiplicative operators* on page 6-188

*6.22 String manipulation operators* on page 6-189

*6.23 Shift operators* on page 6-190

*6.24 Addition, subtraction, and logical operators* on page 6-191

*6.25 Relational operators* on page 6-192

*6.26 Boolean operators* on page 6-193

## 6.21 Multiplicative operators

Multiplicative operators have the highest precedence of all binary operators. They act only on numeric expressions.

The following table shows the multiplicative operators:

**Table 6-3 Multiplicative operators**

| Operator | Alias | Usage | Explanation |
|---|---|---|---|
| `*` | | `A*B` | Multiply |
| `/` | | `A/B` | Divide |
| `:MOD:` | `%` | `A:MOD:B` | A modulo B |

You can use the `:MOD:` operator on PC-relative expressions to ensure code is aligned correctly. These alignment checks have the form *`PC-relative`*`:MOD:`*`Constant`*. For example:

```
    AREA x,CODE
    ASSERT ({PC}:MOD:4) == 0
    DCB 1
y   DCB 2
    ASSERT (y:MOD:4) == 1
    ASSERT ({PC}:MOD:4) == 2
    END
```

***Related concepts***

*6.20 Binary operators* on page 6-187

*6.5 Register-relative and PC-relative expressions* on page 6-172

*6.14 Numeric expressions* on page 6-181

***Related reference***

*6.15 Syntax of numeric literals* on page 6-182

## 6.22 String manipulation operators

You can use string manipulation operators to concatenate two strings, or to extract a substring.

The following table shows the string manipulation operators. In `CC`, both `A` and `B` must be strings. In the slicing operators `LEFT` and `RIGHT`:

- `A` must be a string.
- `B` must be a numeric expression.

**Table 6-4 String manipulation operators**

| Operator | Usage | Explanation |
|---|---|---|
| `:CC:` | `A:CC:B` | B concatenated onto the end of A |
| `:LEFT:` | `A:LEFT:B` | The left-most B characters of A |
| `:RIGHT:` | `A:RIGHT:B` | The right-most B characters of A |

***Related concepts***

*6.12 String expressions* on page 6-179

*6.14 Numeric expressions* on page 6-181

## 6.23 Shift operators

Shift operators act on numeric expressions, by shifting or rotating the first operand by the amount specified by the second.

The following table shows the shift operators:

**Table 6-5 Shift operators**

| Operator | Alias | Usage | Explanation |
|---|---|---|---|
| `:ROL:` | | `A:ROL:B` | Rotate A left by B bits |
| `:ROR:` | | `A:ROR:B` | Rotate A right by B bits |
| `:SHL:` | `<<` | `A:SHL:B` | Shift A left by B bits |
| `:SHR:` | `>>` | `A:SHR:B` | Shift A right by B bits |

**Note**

`SHR` is a logical shift and does not propagate the sign bit.

***Related concepts***

*6.20 Binary operators* on page 6-187

## 6.24 Addition, subtraction, and logical operators

Addition, subtraction, and logical operators act on numeric expressions.

Logical operations are performed bitwise, that is, independently on each bit of the operands to produce the result.

The following table shows the addition, subtraction, and logical operators:

**Table 6-6 Addition, subtraction, and logical operators**

| Operator | Alias | Usage | Explanation |
|---|---|---|---|
| `+` | | `A+B` | Add A to B |
| `-` | | `A-B` | Subtract B from A |
| `:AND:` | `&` | `A:AND:B` | Bitwise AND of A and B |
| `:EOR:` | `^` | `A:EOR:B` | Bitwise Exclusive OR of A and B |
| `:OR:` | | `A:OR:B` | Bitwise OR of A and B |

The use of `|` as an alias for `:OR:` is deprecated.

***Related concepts***

*6.20 Binary operators* on page 6-187

## 6.25 Relational operators

Relational operators act on two operands of the same type to produce a logical value.

The operands can be one of:

- Numeric.
- PC-relative.
- Register-relative.
- Strings.

Strings are sorted using ASCII ordering. String `A` is less than string `B` if it is a leading substring of string `B`, or if the left-most character in which the two strings differ is less in string `A` than in string `B`.

Arithmetic values are unsigned, so the value of `0>-1` is `{FALSE}`.

The following table shows the relational operators:

**Table 6-7 Relational operators**

| Operator | Alias | Usage | Explanation |
|---|---|---|---|
| `=` | `==` | `A=B` | A equal to B |
| `>` | | `A>B` | A greater than B |
| `>=` | | `A>=B` | A greater than or equal to B |
| `<` | | `A<B` | A less than B |
| `<=` | | `A<=B` | A less than or equal to B |
| `/=` | `<> !=` | `A/=B` | A not equal to B |

***Related concepts***

*6.20 Binary operators* on page 6-187

## 6.26 Boolean operators

Boolean operators perform standard logical operations on their operands. They have the lowest precedence of all operators.

In all three cases, both A and B must be expressions that evaluate to either `{TRUE}` or `{FALSE}`.

The following table shows the Boolean operators:

**Table 6-8 Boolean operators**

| Operator | Alias | Usage | Explanation |
|---|---|---|---|
| `:LAND:` | `&&` | `A:LAND:B` | Logical AND of A and B |
| `:LEOR:` | | `A:LEOR:B` | Logical Exclusive OR of A and B |
| `:LOR:` | `\|\|` | `A:LOR:B` | Logical OR of A and B |

***Related concepts***

*6.20 Binary operators* on page 6-187

## 6.27 Operator precedence

`armasm` includes an extensive set of operators for use in expressions. It evaluates them using a strict order of precedence.

Many of the operators resemble their counterparts in high-level languages such as C.

`armasm` evaluates operators in the following order:

1. Expressions in parentheses are evaluated first.
2. Operators are applied in precedence order.
3. Adjacent unary operators are evaluated from right to left.
4. Binary operators of equal precedence are evaluated from left to right.

***Related concepts***

*6.19 Unary operators* on page 6-186

*6.20 Binary operators* on page 6-187

*6.28 Difference between operator precedence in assembly language and C* on page 6-195

***Related reference***

*6.21 Multiplicative operators* on page 6-188

*6.22 String manipulation operators* on page 6-189

*6.23 Shift operators* on page 6-190

*6.24 Addition, subtraction, and logical operators* on page 6-191

*6.25 Relational operators* on page 6-192

*6.26 Boolean operators* on page 6-193

## 6.28 Difference between operator precedence in assembly language and C

`armasm` does not follow exactly the same order of precedence when evaluating operators as a C compiler.

For example, `(1 + 2 :SHR: 3)` evaluates as `(1 + (2 :SHR: 3))` = 1 in assembly language. The equivalent expression in C evaluates as `((1 + 2) >> 3)` = 0.

Arm recommends you use brackets to make the precedence explicit.

If your code contains an expression that would parse differently in C, and you are not using the `--unsafe` option, `armasm` gives a warning:

```
A1466W: Operator precedence means that expression would evaluate differently in C
```

In the following tables:
- The highest precedence operators are at the top of the list.
- The highest precedence operators are evaluated first.
- Operators of equal precedence are evaluated from left to right.

The following table shows the order of precedence of operators in assembly language, and a comparison with the order in C.

**Table 6-9 Operator precedence in Arm assembly language**

| assembly language precedence | equivalent C operators |
|---|---|
| unary operators | unary operators |
| `* / :MOD:` | `* / %` |
| string manipulation | n/a |
| `:SHL: :SHR: :ROR: :ROL:` | `<< >>` |
| `+ - :AND: :OR: :EOR:` | `+ - & \| ^` |
| `= > >= < <= /= <>` | `== > >= < <= !=` |
| `:LAND: :LOR: :LEOR:` | `&& \|\|` |

The following table shows the order of precedence of operators in C.

**Table 6-10 Operator precedence in C**

| C precedence |
|---|
| unary operators |
| `* / %` |
| `+ -` (as binary operators) |
| `<< >>` |
| `< <= > >=` |
| `== !=` |
| `&` |
| `^` |
| `\|` |
| `&&` |
| `\|\|` |

***Related concepts***

*6.20 Binary operators* on page 6-187

***Related reference***

*6.27 Operator precedence* on page 6-194

# Chapter 7
# Directives Reference

Describes the directives that are provided by the Arm assembler, `armasm`.

It contains the following sections:

## 7.1 Alphabetical list of directives

The Arm assembler, armasm, provides various directives.

The following table lists them:

**Table 7-1 List of directives**

| Directive | Directive | Directive |
|---|---|---|
| ALIAS | EQU | LTORG |
| ALIGN | EXPORT or GLOBAL | MACRO and MEND |
| ARM or CODE32 | EXPORTAS | MAP |
| AREA | EXTERN | MEND (see MACRO) |
| ASSERT | FIELD | MEXIT |
| ATTR | FRAME ADDRESS | NOFP |
| CN | FRAME POP | OPT |
| CODE16 | FRAME PUSH | PRESERVE8 (see REQUIRE8) |
| COMMON | FRAME REGISTER | PROC see FUNCTION |
| CP | FRAME RESTORE | |
| DATA | FRAME SAVE | RELOC |
| DCB | FRAME STATE REMEMBER | REQUIRE |
| DCD and DCDU | FRAME STATE RESTORE | REQUIRE8 and PRESERVE8 |
| DCDO | FRAME UNWIND ON or OFF | RLIST |
| DCFD and DCFDU | FUNCTION or PROC | RN |
| DCFS and DCFSU | GBLA, GBLL, and GBLS | ROUT |
| DCI | GET or INCLUDE | SETA, SETL, and SETS |
| DCQ and DCQU | GLOBAL (see EXPORT) | SN |
| DCW and DCWU | IF, ELSE, ENDIF, and ELIF | SPACE or FILL |
| DN | IMPORT | SUBT |
| ELIF, ELSE (see IF) | INCBIN | THUMB |
| END | INCLUDE see GET | TTL |
| ENDFUNC or ENDP | INFO | WHILE and WEND |
| ENDIF (see IF) | KEEP | WN and XN |
| ENTRY | LCLA, LCLL, and LCLS | |

## 7.2 About assembly control directives

Some assembler directives control conditional assembly, looping, inclusions, and macros.

These directives are as follows:

- `MACRO` and `MEND`.
- `MEXIT`.
- `IF`, `ELSE`, `ENDIF`, and `ELIF`.
- `WHILE` and `WEND`.

### Nesting directives

The following structures can be nested to a total depth of 256:

- `MACRO` definitions.
- `WHILE...WEND` loops.
- `IF...ELSE...ENDIF` conditional structures.
- `INCLUDE` file inclusions.

The limit applies to all structures taken together, regardless of how they are nested. The limit is not 256 of each type of structure.

***Related reference***

*7.51 MACRO and MEND* on page 7-256
*7.53 MEXIT* on page 7-260
*7.44 IF, ELSE, ENDIF, and ELIF* on page 7-247
*7.67 WHILE and WEND* on page 7-277

## 7.3 About frame directives

Frame directives enable debugging and profiling of assembly language functions. They also enable the stack usage of functions to be calculated.

Correct use of these directives:

- Enables the `armlink --callgraph` option to calculate stack usage of assembler functions.

  The following are the rules that determine stack usage:
  - If a function is not marked with `PROC` or `ENDP`, stack usage is unknown.
  - If a function is marked with `PROC` or `ENDP` but with no `FRAME PUSH` or `FRAME POP`, stack usage is assumed to be zero. This means that there is no requirement to manually add `FRAME PUSH 0` or `FRAME POP 0`.
  - If a function is marked with `PROC` or `ENDP` and with `FRAME PUSH n` or `FRAME POP n`, stack usage is assumed to be `n` bytes.
- Helps you to avoid errors in function construction, particularly when you are modifying existing code.
- Enables the assembler to alert you to errors in function construction.
- Enables backtracing of function calls during debugging.
- Enables the debugger to profile assembler functions.

If you require profiling of assembler functions, but do not want frame description directives for other purposes:

- You must use the `FUNCTION` and `ENDFUNC`, or `PROC` and `ENDP`, directives.
- You can omit the other `FRAME` directives.
- You only have to use the `FUNCTION` and `ENDFUNC` directives for the functions you want to profile.

In DWARF, the canonical frame address is an address on the stack specifying where the call frame of an interrupted function is located.

***Related reference***

*7.30 FRAME ADDRESS* on page 7-233
*7.31 FRAME POP* on page 7-234
*7.32 FRAME PUSH* on page 7-235
*7.33 FRAME REGISTER* on page 7-236
*7.34 FRAME RESTORE* on page 7-237
*7.35 FRAME RETURN ADDRESS* on page 7-238
*7.36 FRAME SAVE* on page 7-239
*7.37 FRAME STATE REMEMBER* on page 7-240
*7.38 FRAME STATE RESTORE* on page 7-241
*7.39 FRAME UNWIND ON* on page 7-242
*7.40 FRAME UNWIND OFF* on page 7-243
*7.41 FUNCTION or PROC* on page 7-244
*7.24 ENDFUNC or ENDP* on page 7-226

## 7.4 ALIAS

The `ALIAS` directive creates an alias for a symbol.

### Syntax

`ALIAS` *`name, aliasname`*

where:

*`name`*

is the name of the symbol to create an alias for.

*`aliasname`*

is the name of the alias to be created.

### Usage

The symbol *`name`* must already be defined in the source file before creating an alias for it. Properties of *`name`* set by the `EXPORT` directive are not inherited by *`aliasname`*, so you must use `EXPORT` on *`aliasname`* if you want to make the alias available outside the current source file. Apart from the properties set by the `EXPORT` directive, *`name`* and *`aliasname`* are identical.

### Correct example

```
baz
bar PROC
    BX lr
    ENDP
    ALIAS  bar,foo    ; foo is an alias for bar
    EXPORT bar
    EXPORT foo        ; foo and bar have identical properties
                      ; because foo was created using ALIAS
    EXPORT baz        ; baz and bar are not identical
                      ; because the size field of baz is not set
```

### Incorrect example

```
    EXPORT bar
    IMPORT car
    ALIAS  bar,foo ; ERROR - bar is not defined yet
    ALIAS  car,boo ; ERROR - car is external
bar PROC
    BX lr
    ENDP
```

***Related reference***

*7.27 EXPORT or GLOBAL* on page 7-229

## 7.5 ALIGN

The `ALIGN` directive aligns the current location to a specified boundary by padding with zeros or `NOP` instructions.

### Syntax

`ALIGN {`*`expr`*`{,`*`offset`*`{,`*`pad`*`{,`*`padsize`*`}}}}`

where:

*`expr`*

is a numeric expression evaluating to any power of 2 from $2^0$ to $2^{31}$

*`offset`*

can be any numeric expression

*`pad`*

can be any numeric expression

*`padsize`*

can be 1, 2 or 4.

### Operation

The current location is aligned to the next lowest address of the form:

*`offset + n * expr`*

*`n`* is any integer which the assembler selects to minimise padding.

If *`expr`* is not specified, `ALIGN` sets the current location to the next word (four byte) boundary. The unused space between the previous and the new current location are filled with:

- Copies of *`pad`*, if *`pad`* is specified.
- `NOP` instructions, if all the following conditions are satisfied:
  - *`pad`* is not specified.
  - The `ALIGN` directive follows A32 or T32 instructions.
  - The current section has the `CODEALIGN` attribute set on the `AREA` directive.
- Zeros otherwise.

*`pad`* is treated as a byte, halfword, or word, according to the value of *`padsize`*. If *`padsize`* is not specified, *`pad`* defaults to bytes in data sections, halfwords in T32 code, or words in A32 code.

### Usage

Use `ALIGN` to ensure that your data and code is aligned to appropriate boundaries. This is typically required in the following circumstances:

- The `ADR` T32 pseudo-instruction can only load addresses that are word aligned, but a label within T32 code might not be word aligned. Use `ALIGN 4` to ensure four-byte alignment of an address within T32 code.
- Use `ALIGN` to take advantage of caches on some Arm processors. For example, the Arm940T™ processor has a cache with 16-byte lines. Use `ALIGN 16` to align function entries on 16-byte boundaries and maximize the efficiency of the cache.
- A label on a line by itself can be arbitrarily aligned. Following A32 code is word-aligned (T32 code is halfword aligned). The label therefore does not address the code correctly. Use `ALIGN 4` (or `ALIGN 2` for T32) before the label.

Alignment is relative to the start of the ELF section where the routine is located. The section must be aligned to the same, or coarser, boundaries. The `ALIGN` attribute on the `AREA` directive is specified differently.

### Examples

```
        AREA    cacheable, CODE, ALIGN=3
rout1   ; code         ; aligned on 8-byte boundary
        ; code
        MOV     pc,lr  ; aligned only on 4-byte boundary
        ALIGN   8      ; now aligned on 8-byte boundary
rout2   ; code
```

In the following example, the `ALIGN` directive tells the assembler that the next instruction is word aligned and offset by 3 bytes. The 3 byte offset is counted from the previous word aligned address, resulting in the second `DCB` placed in the last byte of the same word and 2 bytes of padding are to be added.

```
        AREA    OffsetExample, CODE
        DCB     1      ; This example places the two bytes in the first
        ALIGN   4,3    ; and fourth bytes of the same word.
        DCB     1      ; The second DCB is offset by 3 bytes from the
                       ; first DCB.
```

In the following example, the `ALIGN` directive tells the assembler that the next instruction is word aligned and offset by 2 bytes. Here, the 2 byte offset is counted from the next word aligned address, so the value *n* is set to 1 (*n*=0 clashes with the third `DCB`). This time three bytes of padding are to be added.

```
        AREA    OffsetExample1, CODE
        DCB     1      ; In this example, n cannot be 0 because it
        DCB     1      ; clashes with the 3rd DCB. The assembler
        DCB     1      ; sets n to 1.
        ALIGN   4,2    ; The next instruction is word aligned and
        DCB     2      ; offset by 2.
```

In the following example, the `DCB` directive makes the PC misaligned. The `ALIGN` directive ensures that the label `subroutine1` and the following instruction are word aligned.

```
        AREA    Example, CODE, READONLY
start   LDR     r6,=label1
        ; code
        MOV     pc,lr
label1  DCB     1      ; PC now misaligned
        ALIGN          ; ensures that subroutine1 addresses
subroutine1            ; the following instruction.
        MOV r5,#0x5
```

***Related reference***

*7.6 AREA* on page 7-205

## 7.6 AREA

The `AREA` directive instructs the assembler to assemble a new code or data section.

### Syntax

`AREA` *`sectionname`*`{,`*`attr`*`}{,`*`attr`*`}...`

where:

*`sectionname`*

is the name to give to the section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.

You can choose any name for your sections. However, names starting with a non-alphabetic character must be enclosed in bars or a missing section name error is generated. For example, `|1_DataArea|`.

Certain names are conventional. For example, `|.text|` is used for code sections produced by the C compiler, or for code sections otherwise associated with the C library.

*`attr`*

are one or more comma-delimited section attributes. Valid attributes are:

`ALIGN=`*`expression`*

By default, ELF sections are aligned on a four-byte boundary. *`expression`* can have any integer value from 0 to 31. The section is aligned on a $2^{expression}$-byte boundary. For example, if *expression* is 10, the section is aligned on a 1KB boundary.

This is not the same as the way that the `ALIGN` directive is specified.

**Note**

Do not use `ALIGN=0` or `ALIGN=1` for A32 code sections.

Do not use `ALIGN=0` for T32 code sections.

`ASSOC=`*`section`*

*`section`* specifies an associated ELF section. *`sectionname`* must be included in any link that includes *`section`*

`CODE`

Contains machine instructions. `READONLY` is the default.

`CODEALIGN`

Causes `armasm` to insert `NOP` instructions when the `ALIGN` directive is used after A32 or T32 instructions within the section, unless the `ALIGN` directive specifies a different padding. `CODEALIGN` is the default for execute-only sections.

`COMDEF`

> **Note**
>
> This attribute is deprecated. Use the `COMGROUP` attribute.

Is a common section definition. This ELF section can contain code or data. It must be identical to any other section of the same name in other source files.

Identical ELF sections with the same name are overlaid in the same section of memory by the linker. If any are different, the linker generates a warning and does not overlay the sections.

`COMGROUP=`*`symbol_name`*

Is the signature that makes the `AREA` part of the named ELF section group. See the GROUP=*`symbol_name`* for more information. The `COMGROUP` attribute marks the ELF section group with the `GRP_COMDAT` flag.

`COMMON`

Is a common data section. You must not define any code or data in it. It is initialized to zeros by the linker. All common sections with the same name are overlaid in the same section of memory by the linker. They do not all have to be the same size. The linker allocates as much space as is required by the largest common section of each name.

`DATA`

Contains data, not instructions. `READWRITE` is the default.

`EXECONLY`

Indicates that the section is execute-only. Execute-only sections must also have the `CODE` attribute, and must not have any of the following attributes:

- `READONLY.`
- `READWRITE.`
- `DATA.`
- `ZEROALIGN.`

`armasm` faults if any of the following occur in an execute-only section:

- Explicit data definitions, for example `DCD` and `DCB`.
- Implicit data definitions, for example `LDR r0, =0xaabbccdd`.
- Literal pool directives, for example `LTORG`, if there is literal data to be emitted.
- `INCBIN` or `SPACE` directives.
- `ALIGN` directives, if the required alignment cannot be accomplished by padding with `NOP` instructions. `armasm` implicitly applies the `CODEALIGN` attribute to sections with the `EXECONLY` attribute.

`FINI_ARRAY`

Sets the ELF type of the current area to `SHT_FINI_ARRAY`.

`GROUP=`*`symbol_name`*

Is the signature that makes the `AREA` part of the named ELF section group. It must be defined by the source file, or a file included by the source file. All `AREAS` with the same *`symbol_name`* signature are part of the same group. Sections within a group are kept or discarded together.

`INIT_ARRAY`

Sets the ELF type of the current area to `SHT_INIT_ARRAY`.

`LINKORDER=`*`section`*

Specifies a relative location for the current section in the image. It ensures that the order of all the sections with the `LINKORDER` attribute, with respect to each other, is the same as the order of the corresponding named *`sections`* in the image.

`MERGE=`*`n`*

Indicates that the linker can merge the current section with other sections with the MERGE=*n* attribute. *n* is the size of the elements in the section, for example *n* is 1 for characters. You must not assume that the section is merged, because the attribute does not force the linker to merge the sections.

`NOALLOC`

Indicates that no memory on the target system is allocated to this area.

`NOINIT`

Indicates that the data section is uninitialized, or initialized to zero. It contains only space reservation directives `SPACE` or `DCB`, `DCD`, `DCDU`, `DCQ`, `DCQU`, `DCW`, or `DCWU` with initialized values of zero. You can decide at link time whether an area is uninitialized or zero-initialized.

**Note**

Arm Compiler does not support systems with ECC or parity protection where the memory is not initialized.

`PREINIT_ARRAY`

Sets the ELF type of the current area to `SHT_PREINIT_ARRAY`.

`READONLY`

Indicates that this section must not be written to. This is the default for Code areas.

`READWRITE`

Indicates that this section can be read from and written to. This is the default for Data areas.

`SECFLAGS=`*`n`*

Adds one or more ELF flags, denoted by *n*, to the current section.

`SECTYPE=`*`n`*

Sets the ELF type of the current section to *n*.

`STRINGS`

Adds the `SHF_STRINGS` flag to the current section. To use the `STRINGS` attribute, you must also use the `MERGE=1` attribute. The contents of the section must be strings that are nul-terminated using the `DCB` directive.

`ZEROALIGN`

Causes `armasm` to insert zeros when the `ALIGN` directive is used after A32 or T32 instructions within the section, unless the `ALIGN` directive specifies a different padding. `ZEROALIGN` is the default for sections that are not execute-only.

## Usage

Use the `AREA` directive to subdivide your source file into ELF sections. You can use the same name in more than one `AREA` directive. All areas with the same name are placed in the same ELF section. Only the attributes of the first `AREA` directive of a particular name are applied.

In general, Arm recommends that you use separate ELF sections for code and data. However, you can put data in code sections. Large programs can usually be conveniently divided into several code sections. Large independent data sets are also usually best placed in separate sections.

The scope of numeric local labels is defined by `AREA` directives, optionally subdivided by `ROUT` directives.

There must be at least one `AREA` directive for an assembly.

**Note**

`armasm` emits `R_ARM_TARGET1` relocations for the `DCD` and `DCDU` directives if the directive uses PC-relative expressions and is in any of the `PREINIT_ARRAY`, `FINI_ARRAY`, or `INIT_ARRAY` ELF sections. You can override the relocation using the `RELOC` directive after each `DCD` or `DCDU` directive. If this relocation is used, read-write sections might become read-only sections at link time if the platform ABI permits this.

---

## Example

The following example defines a read-only code section named `Example`:

```
    AREA    Example,CODE,READONLY   ; An example code section.
            ; code
```

***Related concepts***

*2.3 ELF sections and the AREA directive* on page 2-31

***Related reference***

*7.5 ALIGN* on page 7-203

*7.57 RELOC* on page 7-266

*7.16 DCD and DCDU* on page 7-218

***Related information***

*Information about image structure and generation*

## 7.7 ARM or CODE32 directive

The `ARM` directive instructs the assembler to interpret subsequent instructions as A32 instructions, using either the UAL or the pre-UAL Arm assembler language syntax. `CODE32` is a synonym for `ARM`.

——— **Note** ———

Not supported for AArch64 state.

———————————

### Syntax

`ARM`

### Usage

In files that contain code using different instruction sets, the `ARM` directive must precede any A32 code.

If necessary, this directive also inserts up to three bytes of padding to align to the next word boundary.

This directive does not assemble to any instructions. It also does not change the state. It only instructs `armasm` to assemble A32 instructions as appropriate, and inserts padding if necessary.

### Example

This example shows how you can use `ARM` and `THUMB` directives to switch state and assemble both A32 and T32 instructions in a single area.

```
        AREA ToT32, CODE, READONLY      ; Name this block of code
        ENTRY                           ; Mark first instruction to execute
        ARM                             ; Subsequent instructions are A32
start
        ADR     r0, into_t32 + 1        ; Processor starts in A32 state
        BX      r0                      ; Inline switch to T32 state
        THUMB                           ; Subsequent instructions are T32
into_t32
        MOVS    r0, #10                 ; New-style T32 instructions
```

***Related reference***

*7.11 CODE16 directive* on page 7-213

*7.65 THUMB directive* on page 7-275

***Related information***

*Arm Architecture Reference Manual*

## 7.8 ASSERT

The `ASSERT` directive generates an error message during assembly if a given assertion is false.

### Syntax

`ASSERT` *`logical-expression`*

where:

*`logical-expression`*

is an assertion that can evaluate to either `{TRUE}` or `{FALSE}`.

### Usage

Use `ASSERT` to ensure that any necessary condition is met during assembly.

If the assertion is false an error message is generated and assembly fails.

### Example

```
        ASSERT  label1 <= label2    ; Tests if the address
                                    ; represented by label1
                                    ; is <= the address
                                    ; represented by label2.
```

***Related reference***

*7.47 INFO* on page 7-252

## 7.9 ATTR

The `ATTR` set directives set values for the ABI build attributes. The `ATTR` scope directives specify the scope for which the set value applies to.

### Syntax

`ATTR FILESCOPE`

`ATTR SCOPE` *`name`*

`ATTR` *`settype tagid, value`*

where:

*`name`*

is a section name or symbol name.

*`settype`*

can be any of:

- `SETVALUE`.
- `SETSTRING`.
- `SETCOMPATWITHVALUE`.
- `SETCOMPATWITHSTRING`.

*`tagid`*

is an attribute tag name (or its numerical value) defined in the ABI for the Arm Architecture.

*`value`*

depends on *`settype`*:

- is a 32-bit integer value when *`settype`* is `SETVALUE` or `SETCOMPATWITHVALUE`.
- is a nul-terminated string when *`settype`* is `SETSTRING` or `SETCOMPATWITHSTRING`.

### Usage

The `ATTR` set directives following the `ATTR FILESCOPE` directive apply to the entire object file. The `ATTR` set directives following the `ATTR SCOPE` *`name`* directive apply only to the named section or symbol.

For tags that expect an integer, you must use `SETVALUE` or `SETCOMPATWITHVALUE`. For tags that expect a string, you must use `SETSTRING` or `SETCOMPATWITHSTRING`.

Use `SETCOMPATWITHVALUE` and `SETCOMPATWITHSTRING` to set tag values which the object file is also compatible with.

### Examples

```
    ATTR  SETSTRING Tag_CPU_raw_name, "Cortex-A8"
    ATTR  SETVALUE  Tag_VFP_arch, 3   ; VFPv3 instructions permitted.
    ATTR  SETVALUE  10, 3             ; 10 is the numerical value of
                                      ; Tag_VFP_arch.
```

***Related information***

*Addenda to, and Errata in, the ABI for the Arm Architecture*

## 7.10 CN

The `CN` directive defines a name for a coprocessor register.

### Syntax

*`name`* `CN` *`expr`*

where:

*`name`*

is the name to be defined for the coprocessor register. *`name`* cannot be the same as any of the predefined names.

*`expr`*

evaluates to a coprocessor register number from 0 to 15.

### Usage

Use `CN` to allocate convenient names to registers, to help you remember what you use each register for.

——— **Note** ———

Avoid conflicting uses of the same register under different names.

———————————

The names c0 to c15 are predefined.

### Example

```
power    CN  6        ; defines power as a symbol for
                      ; coprocessor register 6
```

## 7.11 CODE16 directive

The `CODE16` directive instructs the assembler to interpret subsequent instructions as T32 instructions, using the UAL syntax.

—————— **Note** ——————

Not supported for AArch64 state.

————————————————

### Syntax

`CODE16`

### Usage

In files that contain code using different instruction sets,`CODE16` must precede T32 code written in pre-UAL syntax.

If necessary, this directive also inserts one byte of padding to align to the next halfword boundary.

This directive does not assemble to any instructions. It also does not change the state. It only instructs `armasm` to assemble T32 instructions as appropriate, and inserts padding if necessary.

***Related reference***

*7.7 ARM or CODE32 directive* on page 7-209

*7.65 THUMB directive* on page 7-275

## 7.12 COMMON

The `COMMON` directive allocates a block of memory of the defined size, at the specified symbol.

### Syntax

`COMMON` *`symbol`*`{,`*`size`*`{,`*`alignment`*`}} {[`*`attr`*`]}`

where:

*`symbol`*

is the symbol name. The symbol name is case-sensitive.

*`size`*

is the number of bytes to reserve.

*`alignment`*

is the alignment.

*`attr`*

can be any one of:

**`DYNAMIC`**

sets the ELF symbol visibility to `STV_DEFAULT`.

**`PROTECTED`**

sets the ELF symbol visibility to `STV_PROTECTED`.

**`HIDDEN`**

sets the ELF symbol visibility to `STV_HIDDEN`.

**`INTERNAL`**

sets the ELF symbol visibility to `STV_INTERNAL`.

### Usage

You specify how the memory is aligned. If the alignment is omitted, the default alignment is four. If the size is omitted, the default size is zero.

You can access this memory as you would any other memory, but no space is allocated by the assembler in object files. The linker allocates the required space as zero-initialized memory during the link stage.

You cannot define, `IMPORT` or `EXTERN` a symbol that has already been created by the `COMMON` directive. In the same way, if a symbol has already been defined or used with the `IMPORT` or `EXTERN` directive, you cannot use the same symbol for the `COMMON` directive.

### Correct example

```
    LDR       r0, =xyz
    COMMON    xyz,255,4   ; defines 255 bytes of ZI store, word-aligned
```

### Incorrect example

```
    COMMON    foo,4,4
    COMMON    bar,4,4
foo DCD       0         ; cannot define label with same name as COMMON
    IMPORT    bar       ; cannot import label with same name as COMMON
```

## 7.13 CP

The `CP` directive defines a name for a specified coprocessor.

### Syntax

*`name`* `CP` *`expr`*

where:

*`name`*

is the name to be assigned to the coprocessor. *`name`* cannot be the same as any of the predefined names.

*`expr`*

evaluates to a coprocessor number within the range 0 to 15.

### Usage

Use `CP` to allocate convenient names to coprocessors, to help you to remember what you use each one for.

**Note**

Avoid conflicting uses of the same coprocessor under different names.

The names p0 to p15 are predefined for coprocessors 0 to 15.

### Example

```
dmu    CP  6       ; defines dmu as a symbol for
                   ; coprocessor 6
```

## 7.14 DATA

The `DATA` directive is no longer required. It is ignored by the assembler.

## 7.15 DCB

The `DCB` directive allocates one or more bytes of memory, and defines the initial runtime contents of the memory.

### Syntax

`{label} DCB expr{,expr}...`

where:

*`expr`*

is either:

- A numeric expression that evaluates to an integer in the range -128 to 255.
- A quoted string. The characters of the string are loaded into consecutive bytes of store.

### Usage

If `DCB` is followed by an instruction, use an `ALIGN` directive to ensure that the instruction is aligned.

`=` is a synonym for `DCB`.

### Example

Unlike C strings, Arm assembler strings are not nul-terminated. You can construct a nul-terminated C string using `DCB` as follows:

```
C_string   DCB  "C_string",0
```

***Related concepts***

*6.14 Numeric expressions* on page 6-181

***Related reference***

*7.16 DCD and DCDU* on page 7-218

*7.21 DCQ and DCQU* on page 7-223

*7.22 DCW and DCWU* on page 7-224

*7.64 SPACE or FILL* on page 7-274

*7.5 ALIGN* on page 7-203

## 7.16 DCD and DCDU

The `DCD` directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory. `DCDU` is the same, except that the memory alignment is arbitrary.

### Syntax

`{label} DCD{U} expr{,expr}`

where:

*`expr`*

is either:

- A numeric expression.
- A PC-relative expression.

### Usage

`DCD` inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment.

Use `DCDU` if you do not require alignment.

`&` is a synonym for `DCD`.

### Examples

```
data1   DCD     1,5,20      ; Defines 3 words containing
                            ; decimal values 1, 5, and 20
data2   DCD     mem06 + 4   ; Defines 1 word containing 4 +
                            ; the address of the label mem06
        AREA    MyData, DATA, READWRITE
        DCB     255         ; Now misaligned ...
data3   DCDU    1,5,20      ; Defines 3 words containing
                            ; 1, 5 and 20, not word aligned
```

***Related concepts***

*6.14 Numeric expressions* on page 6-181

***Related reference***

*7.15 DCB* on page 7-217

*7.21 DCQ and DCQU* on page 7-223

*7.22 DCW and DCWU* on page 7-224

*7.64 SPACE or FILL* on page 7-274

*7.20 DCI* on page 7-222

## 7.17 DCDO

The `DCDO` directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory as an offset from the *static base register*, sb (R9).

### Syntax

`{label} DCDO expr{,expr}...`

where:

*expr*

is a register-relative expression or label. The base register must be sb.

### Usage

Use `DCDO` to allocate space in memory for static base register relative relocatable addresses.

### Example

```
        IMPORT  externsym
        DCDO    externsym   ; 32-bit word relocated by offset of
                            ; externsym from base of SB section.
```

## 7.18 DCFD and DCFDU

The `DCFD` directive allocates memory for word-aligned double-precision floating-point numbers, and defines the initial runtime contents of the memory. `DCFDU` is the same, except that the memory alignment is arbitrary.

### Syntax

`{label} DCFD{U} fpliteral{,fpliteral}...`

where:

*`fpliteral`*

is a double-precision floating-point literal.

### Usage

Double-precision numbers occupy two words and must be word aligned to be used in arithmetic operations. The assembler inserts up to three bytes of padding before the first defined number, if necessary, to achieve four-byte alignment.

Use `DCFDU` if you do not require alignment.

The word order used when converting *`fpliteral`* to internal form is controlled by the floating-point architecture selected. You cannot use `DCFD` or `DCFDU` if you select the `--fpu none` option.

The range for double-precision numbers is:
- Maximum 1.79769313486231571e+308.
- Minimum 2.22507385850720138e-308.

### Examples

```
        DCFD    1E308,-4E-100
        DCFDU   10000,-.1,3.1E26
```

***Related reference***

*7.19 DCFS and DCFSU* on page 7-221

*6.16 Syntax of floating-point literals* on page 6-183

## 7.19 DCFS and DCFSU

The `DCFS` directive allocates memory for word-aligned single-precision floating-point numbers, and defines the initial runtime contents of the memory. `DCFSU` is the same, except that the memory alignment is arbitrary.

### Syntax

`{`*`label`*`} DCFS{U}`*`fpliteral`*`{,`*`fpliteral`*`}...`

where:

*`fpliteral`*

is a single-precision floating-point literal.

### Usage

Single-precision numbers occupy one word and must be word aligned to be used in arithmetic operations. `DCFS` inserts up to three bytes of padding before the first defined number, if necessary to achieve four-byte alignment.

Use `DCFSU` if you do not require alignment.

The range for single-precision values is:
- Maximum 3.40282347e+38.
- Minimum 1.17549435e-38.

### Examples

```
        DCFS    1E3,-4E-9
        DCFSU   1.0,-.1,3.1E6
```

***Related reference***

*7.18 DCFD and DCFDU* on page 7-220

*6.16 Syntax of floating-point literals* on page 6-183

## 7.20 DCI

The `DCI` directive allocates memory that is aligned and defines the initial runtime contents of the memory.

In A32 code, it allocates one or more words of memory, aligned on four-byte boundaries.

In T32 code, it allocates one or more halfwords of memory, aligned on two-byte boundaries.

### Syntax

`{label} DCI{.W} expr{,expr}`

where:

*expr*
is a numeric expression.

*.W*
if present, indicates that four bytes must be inserted in T32 code.

### Usage

The `DCI` directive is very like the `DCD` or `DCW` directives, but the location is marked as code instead of data. Use `DCI` when writing macros for new instructions not supported by the version of the assembler you are using.

In A32 code, `DCI` inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment. In T32 code, `DCI` inserts an initial byte of padding, if necessary, to achieve two-byte alignment.

You can use `DCI` to insert a bit pattern into the instruction stream. For example, use:

```
DCI 0x46c0
```

to insert the T32 operation `MOV r8,r8`.

### Example macro

```
    MACRO           ; this macro translates newinstr Rd,Rm
                    ; to the appropriate machine code
    newinst     $Rd,$Rm
    DCI         0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm
    MEND
```

### 32-bit T32 example

```
    DCI.W   0xf3af8000   ; inserts 32-bit NOP, 2-byte aligned.
```

***Related concepts***

*6.14 Numeric expressions* on page 6-181

***Related reference***

*7.16 DCD and DCDU* on page 7-218

*7.22 DCW and DCWU* on page 7-224

## 7.21 DCQ and DCQU

The `DCQ` directive allocates one or more eight-byte blocks of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory. `DCQU` is the same, except that the memory alignment is arbitrary.

### Syntax

`{label} DCQ{U} {-}literal{,{-}literal…}`

`{label} DCQ{U} expr{,expr…}`

where:

*`literal`*

is a 64-bit numeric literal.

The range of numbers permitted is 0 to $2^{64}$-1.

In addition to the characters normally permitted in a numeric literal, you can prefix *`literal`* with a minus sign. In this case, the range of numbers permitted is -$2^{63}$ to -1.

The result of specifying *`-n`* is the same as the result of specifying $2^{64}$–*`n`*.

*`expr`*

is either:
- A numeric expression.
- A PC-relative expression.

**Note**

`armasm` accepts expressions in `DCQ` and `DCQU` directives only when you are assembling for AArch64 targets.

---

### Usage

`DCQ` inserts up to three bytes of padding before the first defined eight-byte block, if necessary, to achieve four-byte alignment.

Use `DCQU` if you do not require alignment.

### Correct example

```
        AREA    MiscData, DATA, READWRITE
data    DCQ     -225,2_101     ; 2_101 means binary 101.
```

### Incorrect example

```
number  EQU     2              ; This code assembles for AArch64 targets only.
        DCQU    number         ; For AArch32 targets, DCQ and DCQU only accept
                               ; literals, not expressions.
```

***Related concepts***

*6.14 Numeric expressions* on page 6-181

***Related reference***

*7.15 DCB* on page 7-217

*7.16 DCD and DCDU* on page 7-218

*7.22 DCW and DCWU* on page 7-224

*7.64 SPACE or FILL* on page 7-274

## 7.22 DCW and DCWU

The `DCW` directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory. `DCWU` is the same, except that the memory alignment is arbitrary.

### Syntax

{*label*} DCW{U} *expr*{,*expr*}...

where:

*expr*

is a numeric expression that evaluates to an integer in the range -32768 to 65535.

### Usage

`DCW` inserts a byte of padding before the first defined halfword if necessary to achieve two-byte alignment.

Use `DCWU` if you do not require alignment.

### Examples

```
data    DCW     -225,2*number   ; number must already be defined
        DCWU    number+4
```

***Related concepts***

*6.14 Numeric expressions* on page 6-181

***Related reference***

*7.15 DCB* on page 7-217

*7.16 DCD and DCDU* on page 7-218

*7.21 DCQ and DCQU* on page 7-223

*7.64 SPACE or FILL* on page 7-274

## 7.23 END

The `END` directive informs the assembler that it has reached the end of a source file.

### Syntax

`END`

### Usage

Every assembly language source file must end with `END` on a line by itself.

If the source file has been included in a parent file by a `GET` directive, the assembler returns to the parent file and continues assembly at the first line following the `GET` directive.

If `END` is reached in the top-level source file during the first pass without any errors, the second pass begins.

If `END` is reached in the top-level source file during the second pass, the assembler finishes the assembly and writes the appropriate output.

***Related reference***

*7.43 GET or INCLUDE* on page 7-246

## 7.24 ENDFUNC or ENDP

The `ENDFUNC` directive marks the end of an AAPCS-conforming function. `ENDP` is a synonym for `ENDFUNC`.

***Related reference***

*7.41 FUNCTION or PROC* on page 7-244

## 7.25 ENTRY

The `ENTRY` directive declares an entry point to a program.

### Syntax

`ENTRY`

### Usage

A program must have an entry point. You can specify an entry point in the following ways:

- Using the `ENTRY` directive in assembly language source code.
- Providing a `main()` function in C or C++ source code.
- Using the `armlink --entry` command-line option.

You can declare more than one entry point in a program, although a source file cannot contain more than one `ENTRY` directive. For example, a program could contain multiple assembly language source files, each with an `ENTRY` directive. Or it could contain a C or C++ file with a `main()` function and one or more assembly source files with an `ENTRY` directive.

If the program contains multiple entry points, then you must select one of them. You do this by exporting the symbol for the `ENTRY` directive that you want to use as the entry point, then using the `armlink --entry` option to select the exported symbol.

### Example

```
        AREA   ARMex, CODE, READONLY
        ENTRY      ; Entry point for the application.
        EXPORT ep1 ; Export the symbol so the linker can find it
ep1                ; in the object file.
        ; code
        END
```

When you invoke `armlink`, if other entry points are declared in the program, then you must specify `--entry=ep1`, to select `ep1`.

***Related information***

*Image entry points*

*--entry=location*

## 7.26 EQU

The `EQU` directive gives a symbolic name to a numeric constant, a register-relative value or a PC-relative value.

### Syntax

*`name`* `EQU` *`expr{, type}`*

where:

*`name`*

is the symbolic name to assign to the value.

*`expr`*

is a register-relative address, a PC-relative address, an absolute address, or a 32-bit integer constant.

*`type`*

is optional. *`type`* can be any one of:

- `ARM`.
- `THUMB`.
- `CODE32`.
- `CODE16`.
- `DATA`.

You can use *`type`* only if *`expr`* is an absolute address. If *`name`* is exported, the *`name`* entry in the symbol table in the object file is marked as `ARM`, `THUMB`, `CODE32`, `CODE16`, or `DATA`, according to *`type`*. This can be used by the linker.

### Usage

Use `EQU` to define constants. This is similar to the use of **`#define`** to define a constant in C.

* is a synonym for `EQU`.

### Examples

```
abc EQU 2                ; Assigns the value 2 to the symbol abc.
xyz EQU label+8          ; Assigns the address (label+8) to the
                         ; symbol xyz.
fiq EQU 0x1C, CODE32     ; Assigns the absolute address 0x1C to
                         ; the symbol fiq, and marks it as code.
```

***Related reference***

*7.48 KEEP* on page 7-253

*7.27 EXPORT or GLOBAL* on page 7-229

## 7.27 EXPORT or GLOBAL

The `EXPORT` directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. `GLOBAL` is a synonym for `EXPORT`.

### Syntax

`EXPORT {[WEAK]}`

`EXPORT` *`symbol`* `{[SIZE=`*`n`*`]}`

`EXPORT` *`symbol`* `{[`*`type`*`{,`*`set`*`}]}`

`EXPORT` *`symbol`* `[`*`attr`*`{,`*`type`*`{,`*`set`*`}}{,SIZE=`*`n`*`}]`

`EXPORT` *`symbol`* `[WEAK {,`*`attr`*`}{,`*`type`*`{,`*`set`*`}}{,SIZE=`*`n`*`}]`

where:

*`symbol`*

is the symbol name to export. The symbol name is case-sensitive. If *`symbol`* is omitted, all symbols are exported.

**`WEAK`**

*`symbol`* is only imported into other sources if no other source exports an alternative *`symbol`*. If `[WEAK]` is used without *`symbol`*, all exported symbols are weak.

*`attr`*

can be any one of:

**`DYNAMIC`**

sets the ELF symbol visibility to `STV_DEFAULT`.

**`PROTECTED`**

sets the ELF symbol visibility to `STV_PROTECTED`.

**`HIDDEN`**

sets the ELF symbol visibility to `STV_HIDDEN`.

**`INTERNAL`**

sets the ELF symbol visibility to `STV_INTERNAL`.

*`type`*

specifies the symbol type:

**`DATA`**

*`symbol`* is treated as data when the source is assembled and linked.

**`CODE`**

*`symbol`* is treated as code when the source is assembled and linked.

`ELFTYPE=`*`n`*

*`symbol`* is treated as a particular ELF symbol, as specified by the value of *`n`*, where *`n`* can be any number from `0` to `15`.

If unspecified, the assembler determines the most appropriate type. Usually the assembler determines the correct type so you are not required to specify it.

*`set`*

specifies the instruction set:

**`ARM`**

*`symbol`* is treated as an A32 symbol.

**`THUMB`**

*`symbol`* is treated as a T32 symbol.

If unspecified, the assembler determines the most appropriate set.

*`n`*

specifies the size and can be any 32-bit value. If the `SIZE` attribute is not specified, the assembler calculates the size:

- For `PROC` and `FUNCTION` symbols, the size is set to the size of the code until its `ENDP` or `ENDFUNC`.
- For other symbols, the size is the size of instruction or data on the same source line. If there is no instruction or data, the size is zero.

## Usage

Use `EXPORT` to give code in other files access to symbols in the current file.

Use the `[WEAK]` attribute to inform the linker that a different instance of *`symbol`* takes precedence over this one, if a different one is available from another source. You can use the `[WEAK]` attribute with any of the symbol visibility attributes.

## Examples

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external modules.
DoAdd   ADD     r0,r0,r1
```

Symbol visibility can be overridden for duplicate exports. In the following example, the last `EXPORT` takes precedence for both binding and visibility:

```
        EXPORT  SymA[WEAK]      ; Export as weak-hidden
        EXPORT  SymA[DYNAMIC]   ; SymA becomes non-weak dynamic.
```

The following examples show the use of the `SIZE` attribute:

```
        EXPORT symA [SIZE=4]
        EXPORT symA [DATA, SIZE=4]
```

***Related reference***

*7.45 IMPORT and EXTERN* on page 7-249

***Related information***

*ELF for the Arm Architecture*

## 7.28 EXPORTAS

The `EXPORTAS` directive enables you to export a symbol from the object file, corresponding to a different symbol in the source file.

### Syntax

`EXPORTAS` *`symbol1, symbol2`*

where:

*`symbol1`*

is the symbol name in the source file. *`symbol1`* must have been defined already. It can be any symbol, including an area name, a label, or a constant.

*`symbol2`*

is the symbol name you want to appear in the object file.

The symbol names are case-sensitive.

### Usage

Use `EXPORTAS` to change a symbol in the object file without having to change every instance in the source file.

### Examples

```
    AREA data1, DATA       ; Starts a new area data1.
    AREA data2, DATA       ; Starts a new area data2.
    EXPORTAS data2, data1  ; The section symbol referred to as data2
                           ; appears in the object file string table as data1.
one EQU  2
    EXPORTAS one, two      ; The symbol 'two' appears in the object
    EXPORT one             ; file's symbol table with the value 2.
```

***Related reference***

*7.27 EXPORT or GLOBAL* on page 7-229

## 7.29 FIELD

The FIELD directive describes space within a storage map that has been defined using the MAP directive.

### Syntax

{*label*} FIELD *expr*

where:

*label*

is an optional label. If specified, *label* is assigned the value of the storage location counter, {VAR}. The storage location counter is then incremented by the value of *expr*.

*expr*

is an expression that evaluates to the number of bytes to increment the storage counter.

### Usage

If a storage map is set by a MAP directive that specifies a *base-register*, the base register is implicit in all labels defined by following FIELD directives, until the next MAP directive. These register-relative labels can be quoted in load and store instructions.

# is a synonym for FIELD.

### Examples

The following example shows how register-relative labels are defined using the MAP and FIELD directives:

```
    MAP     0,r9        ; set {VAR} to the address stored in R9
    FIELD   4           ; increment {VAR} by 4 bytes
Lab FIELD   4           ; set Lab to the address [R9 + 4]
                        ; and then increment {VAR} by 4 bytes
    LDR     r0,Lab      ; equivalent to LDR r0,[r9,#4]
```

When using the MAP and FIELD directives, you must ensure that the values are consistent in both passes. The following example shows a use of MAP and FIELD that causes inconsistent values for the symbol x. In the first pass sym is not defined, so x is at 0x04+R9. In the second pass, sym is defined, so x is at 0x00+R0. This example results in an assembly error.

```
    MAP 0, r0
    if :LNOT: :DEF: sym
      MAP 0, r9
      FIELD 4  ; x is at 0x04+R9 in first pass
    ENDIF
x   FIELD 4    ; x is at 0x00+R0 in second pass
sym LDR r0, x  ; inconsistent values for x results in assembly error
```

***Related concepts***

*1.3 How the assembler works* on page 1-19

***Related reference***

*7.52 MAP* on page 7-259

*1.4 Directives that can be omitted in pass 2 of the assembler* on page 1-21

## 7.30 FRAME ADDRESS

The `FRAME ADDRESS` directive describes how to calculate the canonical frame address for the following instructions.

### Syntax

`FRAME ADDRESS` *`reg{,offset}`*

where:

*`reg`*

is the register on which the canonical frame address is to be based. This is SP unless the function uses a separate frame pointer.

*`offset`*

is the offset of the canonical frame address from *`reg`*. If *`offset`* is zero, you can omit it.

### Usage

Use `FRAME ADDRESS` if your code alters which register the canonical frame address is based on, or if it changes the offset of the canonical frame address from the register. You must use `FRAME ADDRESS` immediately after the instruction that changes the calculation of the canonical frame address.

You can only use `FRAME ADDRESS` in functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

**Note**

If your code uses a single instruction to save registers and alter the stack pointer, you can use `FRAME PUSH` instead of using both `FRAME ADDRESS` and `FRAME SAVE`.

If your code uses a single instruction to load registers and alter the stack pointer, you can use `FRAME POP` instead of using both `FRAME ADDRESS` and `FRAME RESTORE`.

---

### Example

```
_fn     FUNCTION        ; CFA (Canonical Frame Address) is value
                        ; of SP on entry to function
        PUSH    {r4,fp,ip,lr,pc}
        FRAME PUSH {r4,fp,ip,lr,pc}
        SUB     sp,sp,#4            ; CFA offset now changed
        FRAME ADDRESS sp,24         ; - so we correct it
        ADD     fp,sp,#20
        FRAME ADDRESS fp,4          ; New base register
        ; code using fp to base call-frame on, instead of SP
```

***Related reference***

*7.31 FRAME POP* on page 7-234

*7.32 FRAME PUSH* on page 7-235

## 7.31 FRAME POP

The `FRAME POP` directive informs the assembler when the callee reloads registers.

### Syntax

There are the following alternative syntaxes for `FRAME POP`:

`FRAME POP {`*`reglist`*`}`

`FRAME POP {`*`reglist`*`},`*`n`*

`FRAME POP` *`n`*

where:

*`reglist`*
: is a list of registers restored to the values they had on entry to the function. There must be at least one register in the list.

*`n`*
: is the number of bytes that the stack pointer moves.

### Usage

`FRAME POP` is equivalent to a `FRAME ADDRESS` and a `FRAME RESTORE` directive. You can use it when a single instruction loads registers and alters the stack pointer.

You must use `FRAME POP` immediately after the instruction it refers to.

You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives. You do not have to do this after the last instruction in a function.

If *`n`* is not specified or is zero, the assembler calculates the new offset for the canonical frame address from `{`*`reglist`*`}`. It assumes that:
- Each AArch32 register popped occupies four bytes on the stack.
- Each VFP single-precision register popped occupies four bytes on the stack, plus an extra four-byte word for each list.
- Each VFP double-precision register popped occupies eight bytes on the stack, plus an extra four-byte word for each list.

***Related reference***

*7.30 FRAME ADDRESS* on page 7-233
*7.34 FRAME RESTORE* on page 7-237

## 7.32 FRAME PUSH

The `FRAME PUSH` directive informs the assembler when the callee saves registers, normally at function entry.

### Syntax

There are the following alternative syntaxes for `FRAME PUSH`:

`FRAME PUSH {`*`reglist`*`}`

`FRAME PUSH {`*`reglist`*`},`*`n`*

`FRAME PUSH `*`n`*

where:

*`reglist`*
: is a list of registers stored consecutively below the canonical frame address. There must be at least one register in the list.

*`n`*
: is the number of bytes that the stack pointer moves.

### Usage

`FRAME PUSH` is equivalent to a `FRAME ADDRESS` and a `FRAME SAVE` directive. You can use it when a single instruction saves registers and alters the stack pointer.

You must use `FRAME PUSH` immediately after the instruction it refers to.

You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

If *`n`* is not specified or is zero, the assembler calculates the new offset for the canonical frame address from `{`*`reglist`*`}`. It assumes that:

- Each AArch32 register pushed occupies four bytes on the stack.
- Each VFP single-precision register pushed occupies four bytes on the stack, plus an extra four-byte word for each list.
- Each VFP double-precision register popped occupies eight bytes on the stack, plus an extra four-byte word for each list.

### Example

```
p   PROC ; Canonical frame address is SP + 0
    EXPORT  p
    PUSH    {r4-r6,lr}
         ; SP has moved relative to the canonical frame address,
         ; and registers R4, R5, R6 and LR are now on the stack
    FRAME PUSH {r4-r6,lr}
         ; Equivalent to:
         ; FRAME ADDRESS    sp,16       ; 16 bytes in {R4-R6,LR}
         ; FRAME SAVE    {r4-r6,lr},-16
```

***Related reference***

*7.30 FRAME ADDRESS* on page 7-233

*7.36 FRAME SAVE* on page 7-239

## 7.33 FRAME REGISTER

The `FRAME REGISTER` directive maintains a record of the locations of function arguments held in registers.

### Syntax

`FRAME REGISTER` *`reg1, reg2`*

where:

*`reg1`*

is the register that held the argument on entry to the function.

*`reg2`*

is the register in which the value is preserved.

### Usage

Use the `FRAME REGISTER` directive when you use a register to preserve an argument that was held in a different register on entry to a function.

You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

## 7.34 FRAME RESTORE

The `FRAME RESTORE` directive informs the assembler that the contents of specified registers have been restored to the values they had on entry to the function.

### Syntax

`FRAME RESTORE {`*`reglist`*`}`

where:

*`reglist`*

is a list of registers whose contents have been restored. There must be at least one register in the list.

### Usage

You can only use `FRAME RESTORE` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives. Use it immediately after the callee reloads registers from the stack. You do not have to do this after the last instruction in a function.

*`reglist`* can contain integer registers or floating-point registers, but not both.

**Note**

If your code uses a single instruction to load registers and alter the stack pointer, you can use `FRAME POP` instead of using both `FRAME RESTORE` and `FRAME ADDRESS`.

***Related reference***

*7.31 FRAME POP* on page 7-234

## 7.35 FRAME RETURN ADDRESS

The `FRAME RETURN ADDRESS` directive provides for functions that use a register other than LR for their return address.

### Syntax

`FRAME RETURN ADDRESS` *`reg`*

where:

*`reg`*

is the register used for the return address.

### Usage

Use the `FRAME RETURN ADDRESS` directive in any function that does not use LR for its return address. Otherwise, a debugger cannot backtrace through the function.

You can only use `FRAME RETURN ADDRESS` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives. Use it immediately after the `FUNCTION` or `PROC` directive that introduces the function.

**Note**

Any function that uses a register other than LR for its return address is not AAPCS compliant. Such a function must not be exported.

## 7.36 FRAME SAVE

The `FRAME SAVE` directive describes the location of saved register contents relative to the canonical frame address.

### Syntax

`FRAME SAVE {`*`reglist`*`},` *`offset`*

where:

*`reglist`*

is a list of registers stored consecutively starting at *`offset`* from the canonical frame address. There must be at least one register in the list.

### Usage

You can only use `FRAME SAVE` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Use it immediately after the callee stores registers onto the stack.

*`reglist`* can include registers which are not required for backtracing. The assembler determines which registers it requires to record in the DWARF call frame information.

**Note**

If your code uses a single instruction to save registers and alter the stack pointer, you can use `FRAME PUSH` instead of using both `FRAME SAVE` and `FRAME ADDRESS`.

***Related reference***

*7.32 FRAME PUSH* on page 7-235

## 7.37 FRAME STATE REMEMBER

The `FRAME STATE REMEMBER` directive saves the current information on how to calculate the canonical frame address and locations of saved register values.

### Syntax

`FRAME STATE REMEMBER`

### Usage

During an inline exit sequence the information about calculation of canonical frame address and locations of saved register values can change. After the exit sequence another branch can continue using the same information as before. Use `FRAME STATE REMEMBER` to preserve this information, and `FRAME STATE RESTORE` to restore it.

These directives can be nested. Each `FRAME STATE RESTORE` directive must have a corresponding `FRAME STATE REMEMBER` directive.

You can only use `FRAME STATE REMEMBER` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

### Example

```
        ; function code
        FRAME STATE REMEMBER
            ; save frame state before in-line exit sequence
        POP     {r4-r6,pc}
            ; do not have to FRAME POP here, as control has
            ; transferred out of the function
        FRAME STATE RESTORE
            ; end of exit sequence, so restore state
exitB   ; code for exitB
        POP     {r4-r6,pc}
        ENDP
```

***Related reference***

*7.38 FRAME STATE RESTORE* on page 7-241

*7.41 FUNCTION or PROC* on page 7-244

## 7.38 FRAME STATE RESTORE

The `FRAME STATE RESTORE` directive restores information about how to calculate the canonical frame address and locations of saved register values.

### Syntax

```
FRAME STATE RESTORE
```

### Usage

You can only use `FRAME STATE RESTORE` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

***Related reference***

*7.37 FRAME STATE REMEMBER* on page 7-240
*7.41 FUNCTION or PROC* on page 7-244

## 7.39 FRAME UNWIND ON

The `FRAME UNWIND ON` directive instructs the assembler to produce unwind tables for this and subsequent functions.

### Syntax

```
FRAME UNWIND ON
```

### Usage

You can use this directive outside functions. In this case, the assembler produces unwind tables for all following functions until it reaches a `FRAME UNWIND OFF` directive.

**Note**

A `FRAME UNWIND` directive is not sufficient to turn on exception table generation. Furthermore a `FRAME UNWIND` directive, without other `FRAME` directives, is not sufficient information for the assembler to generate the unwind information.

***Related reference***

*5.26 --exceptions, --no_exceptions* on page 5-124

*5.27 --exceptions_unwind, --no_exceptions_unwind* on page 5-125

## 7.40 FRAME UNWIND OFF

The `FRAME UNWIND OFF` directive instructs the assembler to produce no unwind tables for this and subsequent functions.

### Syntax

```
FRAME UNWIND OFF
```

### Usage

You can use this directive outside functions. In this case, the assembler produces no unwind tables for all following functions until it reaches a `FRAME UNWIND ON` directive.

***Related reference***

*5.26 --exceptions, --no_exceptions* on page 5-124

*5.27 --exceptions_unwind, --no_exceptions_unwind* on page 5-125

## 7.41 FUNCTION or PROC

The `FUNCTION` directive marks the start of a function. `PROC` is a synonym for `FUNCTION`.

### Syntax

`label FUNCTION [{reglist1} [, {reglist2}]]`

where:

*`reglist1`*
: is an optional list of callee-saved AArch32 registers. If *`reglist1`* is not present, and your debugger checks register usage, it assumes that the AAPCS is in use. If you use empty brackets, this informs the debugger that all AArch32 registers are caller-saved.

*`reglist2`*
: is an optional list of callee-saved VFP registers. If you use empty brackets, this informs the debugger that all VFP registers are caller-saved.

### Usage

Use `FUNCTION` to mark the start of functions. The assembler uses `FUNCTION` to identify the start of a function when producing DWARF call frame information for ELF.

`FUNCTION` sets the canonical frame address to be R13 (SP), and the frame state stack to be empty.

Each `FUNCTION` directive must have a matching `ENDFUNC` directive. You must not nest `FUNCTION` and `ENDFUNC` pairs, and they must not contain `PROC` or `ENDP` directives.

You can use the optional *`reglist`* parameters to inform the debugger about an alternative procedure call standard, if you are using your own. Not all debuggers support this feature. See your debugger documentation for details.

If you specify an empty *`reglist`*, using `{}`, this indicates that all registers for the function are caller-saved. Typically you do this when writing a reset vector where the values in all registers are unknown on execution. This avoids problems in a debugger if it tries to construct a backtrace from the values in the registers.

**Note**

`FUNCTION` does not automatically cause alignment to a word boundary (or halfword boundary for T32). Use `ALIGN` if necessary to ensure alignment, otherwise the call frame might not point to the start of the function.

### Examples

```
       ALIGN       ; Ensures alignment.
dadd   FUNCTION    ; Without the ALIGN directive this might not be word-aligned.
       EXPORT  dadd
       PUSH       {r4-r6,lr}    ; This line automatically word-aligned.
       FRAME PUSH {r4-r6,lr}
       ; subroutine body
       POP        {r4-r6,pc}
       ENDFUNC
func6  PROC {r4-r8,r12},{D1-D3} ; Non-AAPCS-conforming function.
       ...
       ENDP
func7  FUNCTION {}  ; Another non-AAPCS-conforming function.
       ...
       ENDFUNC
```

***Related reference***

*7.38 FRAME STATE RESTORE* on page 7-241
*7.30 FRAME ADDRESS* on page 7-233
*7.5 ALIGN* on page 7-203

## 7.42 GBLA, GBLL, and GBLS

The `GBLA`, `GBLL`, and `GBLS` directives declare and initialize global variables.

### Syntax

*`gblx variable`*

where:

*`gblx`*

is one of `GBLA`, `GBLL`, or `GBLS`.

*`variable`*

is the name of the variable. *`variable`* must be unique among symbols within a source file.

### Usage

The `GBLA` directive declares a global arithmetic variable, and initializes its value to 0.

The `GBLL` directive declares a global logical variable, and initializes its value to `{FALSE}`.

The `GBLS` directive declares a global string variable and initializes its value to a null string, `""`.

Using one of these directives for a variable that is already defined re-initializes the variable.

The scope of the variable is limited to the source file that contains it.

Set the value of the variable with a `SETA`, `SETL`, or `SETS` directive.

Global variables can also be set with the `--predefine` assembler command-line option.

### Examples

The following example declares a variable `objectsize`, sets the value of `objectsize` to `0xFF`, and then uses it later in a `SPACE` directive:

```
            GBLA    objectsize     ; declare the variable name
objectsize  SETA    0xFF           ; set its value
            .
            .                      ; other code
            .
            SPACE   objectsize     ; quote the variable
```

The following example shows how to declare and set a variable when you invoke `armasm`. Use this when you want to set the value of a variable at assembly time. `--pd` is a synonym for `--predefine`.

```
armasm --cpu=8-A.32 --predefine "objectsize SETA 0xFF" sourcefile -o objectfile
```

***Related reference***

*7.49 LCLA, LCLL, and LCLS* on page 7-254
*7.63 SETA, SETL, and SETS* on page 7-272
*5.54 --predefine "directive"* on page 5-152

## 7.43 GET or INCLUDE

The `GET` directive includes a file within the file being assembled. The included file is assembled at the location of the `GET` directive. `INCLUDE` is a synonym for `GET`.

### Syntax

`GET` *`filename`*

where:

*`filename`*

is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

### Usage

`GET` is useful for including macro definitions, `EQU`s, and storage maps in an assembly. When assembly of the included file is complete, assembly continues at the line following the `GET` directive.

By default the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the `-i` assembler command line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes ( " " ).

The included file can contain additional `GET` directives to include other files.

If the included file is in a different directory from the current place, this becomes the current place until the end of the included file. The previous current place is then restored.

You cannot use `GET` to include object files.

### Examples

```
    AREA   Example, CODE, READONLY
    GET    file1.s                 ; includes file1 if it exists in the current place
    GET    c:\project\file2.s      ; includes file2
    GET    c:\Program files\file3.s ; space is permitted
```

***Related reference***

*7.46 INCBIN* on page 7-251

*7.2 About assembly control directives* on page 7-200

## 7.44 IF, ELSE, ENDIF, and ELIF

The `IF`, `ELSE`, `ENDIF`, and `ELIF` directives allow you to conditionally assemble sequences of instructions and directives.

### Syntax

```
IF logical-expression
    …;code
{ELSE
    …;code}
ENDIF
```

where:

*logical-expression*

is an expression that evaluates to either `{TRUE}` or `{FALSE}`.

### Usage

Use `IF` with `ENDIF`, and optionally with `ELSE`, for sequences of instructions or directives that are only to be assembled or acted on under a specified condition.

`IF...ENDIF` conditions can be nested.

The `IF` directive introduces a condition that controls whether to assemble a sequence of instructions and directives. `[` is a synonym for `IF`.

The `ELSE` directive marks the beginning of a sequence of instructions or directives that you want to be assembled if the preceding condition fails. `|` is a synonym for `ELSE`.

The `ENDIF` directive marks the end of a sequence of instructions or directives that you want to be conditionally assembled. `]` is a synonym for `ENDIF`.

The `ELIF` directive creates a structure equivalent to `ELSE IF`, without the requirement for nesting or repeating the condition.

### Using ELIF

Without using `ELIF`, you can construct a nested set of conditional instructions like this:

```
IF logical-expression
        instructions
ELSE
    IF logical-expression2
        instructions
    ELSE
        IF logical-expression3
            instructions
        ENDIF
    ENDIF
ENDIF
```

A nested structure like this can be nested up to 256 levels deep.

You can write the same structure more simply using `ELIF`:

```
IF logical-expression
     instructions
ELIF logical-expression2
     instructions
ELIF logical-expression3
     instructions
ENDIF
```

This structure only adds one to the current nesting depth, for the `IF...ENDIF` pair.

### Examples

The following example assembles the first set of instructions if `NEWVERSION` is defined, or the alternative set otherwise:

**Assembly conditional on a variable being defined**

```
        IF :DEF:NEWVERSION
            ; first set of instructions or directives
        ELSE
            ; alternative set of instructions or directives
        ENDIF
```

Invoking `armasm` as follows defines `NEWVERSION`, so the first set of instructions and directives are assembled:

```
armasm --cpu=8-A.32 --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking `armasm` as follows leaves `NEWVERSION` undefined, so the second set of instructions and directives are assembled:

```
armasm --cpu=8-A.32 test.s
```

The following example assembles the first set of instructions if `NEWVERSION` has the value `{TRUE}`, or the alternative set otherwise:

**Assembly conditional on a variable value**

```
        IF NEWVERSION = {TRUE}
            ; first set of instructions or directives
        ELSE
            ; alternative set of instructions or directives
        ENDIF
```

Invoking `armasm` as follows causes the first set of instructions and directives to be assembled:

```
armasm --cpu=8-A.32 --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking `armasm` as follows causes the second set of instructions and directives to be assembled:

```
armasm --cpu=8-A.32 --predefine "NEWVERSION SETL {FALSE}" test.s
```

***Related reference***

*6.25 Relational operators* on page 6-192
*7.2 About assembly control directives* on page 7-200

## 7.45 IMPORT and EXTERN

The `IMPORT` and `EXTERN` directives provide the assembler with a name that is not defined in the current assembly.

### Syntax

`*directive symbol* {[SIZE=*n*]}`

`*directive symbol* {[*type*]}`

`*directive symbol* [*attr*{,*type*}{,SIZE=*n*}]`

`*directive symbol* [WEAK {,*attr*}{,*type*}{,SIZE=*n*}]`

where:

*`directive`*

can be either:

**`IMPORT`**

imports the symbol unconditionally.

**`EXTERN`**

imports the symbol only if it is referred to in the current assembly.

*`symbol`*

is a symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.

**`WEAK`**

prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

*`attr`*

can be any one of:

**`DYNAMIC`**

sets the ELF symbol visibility to `STV_DEFAULT`.

**`PROTECTED`**

sets the ELF symbol visibility to `STV_PROTECTED`.

**`HIDDEN`**

sets the ELF symbol visibility to `STV_HIDDEN`.

**`INTERNAL`**

sets the ELF symbol visibility to `STV_INTERNAL`.

*`type`*

specifies the symbol type:

**`DATA`**

*`symbol`* is treated as data when the source is assembled and linked.

**`CODE`**

*`symbol`* is treated as code when the source is assembled and linked.

`ELFTYPE=`*`n`*

> *`symbol`* is treated as a particular ELF symbol, as specified by the value of *`n`*, where *`n`* can be any number from `0` to `15`.

If unspecified, the linker determines the most appropriate type.

*`n`*

specifies the size and can be any 32-bit value. If the `SIZE` attribute is not specified, the assembler calculates the size:

- For `PROC` and `FUNCTION` symbols, the size is set to the size of the code until its `ENDP` or `ENDFUNC`.
- For other symbols, the size is the size of instruction or data on the same source line. If there is no instruction or data, the size is zero.

## Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If `[WEAK]` is not specified, the linker generates an error if no corresponding symbol is found at link time.

If `[WEAK]` is specified and no corresponding symbol is found at link time:

- If the reference is the destination of a `B` or `BL` instruction, the value of the symbol is taken as the address of the following instruction. This makes the `B` or `BL` instruction effectively a `NOP`.
- Otherwise, the value of the symbol is taken as zero.

## Example

The example tests to see if the C++ library has been linked, and branches conditionally on the result.

```
    AREA    Example, CODE, READONLY
    EXTERN  __CPP_INITIALIZE[WEAK]  ; If C++ library linked, gets the
                                    ; address of __CPP_INITIALIZE
                                    ; function.
    LDR     r0,=__CPP_INITIALIZE    ; If not linked, address is zeroed.
    CMP     r0,#0                   ; Test if zero.
    BEQ     nocplusplus             ; Branch on the result.
```

The following examples show the use of the `SIZE` attribute:

```
    EXTERN symA [SIZE=4]
    EXTERN symA [DATA, SIZE=4]
```

***Related reference***

*7.27 EXPORT or GLOBAL* on page 7-229

***Related information***

*ELF for the Arm Architecture*

## 7.46 INCBIN

The `INCBIN` directive includes a file within the file being assembled. The file is included as it is, without being assembled.

### Syntax

`INCBIN` *`filename`*

where:

*`filename`*

is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

### Usage

You can use `INCBIN` to include data, such as executable files, literals, or any arbitrary data. The contents of the file are added to the current ELF section, byte for byte, without being interpreted in any way. Assembly continues at the line following the `INCBIN` directive.

By default, the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the `-i` assembler command-line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes ( " " ).

### Example

```
    AREA    Example, CODE, READONLY
    INCBIN  file1.dat              ; Includes file1 if it exists in the current place
    INCBIN  c:\project\file2.txt   ; Includes file2.
```

## 7.47 INFO

The `INFO` directive supports diagnostic generation on either pass of the assembly.

### Syntax

`INFO` *`numeric-expression, string-expression{, severity}`*

where:

*`numeric-expression`*

is a numeric expression that is evaluated during assembly. If the expression evaluates to zero:

- No action is taken during pass one.
- *`string-expression`* is printed as a warning during pass two if *`severity`* is 1.
- *`string-expression`* is printed as a message during pass two if *`severity`* is 0 or not specified.

If the expression does not evaluate to zero:

- *`string-expression`* is printed as an error message and the assembly fails irrespective of whether *`severity`* is specified or not (non-zero values for *`severity`* are reserved in this case).

*`string-expression`*

is an expression that evaluates to a string.

*`severity`*

is an optional number that controls the severity of the message. Its value can be either 0 or 1. All other values are reserved.

### Usage

`INFO` provides a flexible means of creating custom error messages.

`!` is very similar to `INFO`, but has less detailed reporting.

### Examples

```
        INFO    0, "Version 1.0"
        IF endofdata <= label1
            INFO    4, "Data overrun at label1"
        ENDIF
```

***Related concepts***

*6.12 String expressions* on page 6-179

*6.14 Numeric expressions* on page 6-181

***Related reference***

*7.8 ASSERT* on page 7-210

## 7.48 KEEP

The `KEEP` directive instructs the assembler to retain named local labels in the symbol table in the object file.

### Syntax

`KEEP {`*`label`*`}`

where:

*`label`*

is the name of the local label to keep. If *`label`* is not specified, all named local labels are kept except register-relative labels.

### Usage

By default, the only labels that the assembler describes in its output object file are:

- Exported labels.
- Labels that are relocated against.

Use `KEEP` to preserve local labels. This can help when debugging. Kept labels appear in the Arm debuggers and in linker map files.

`KEEP` cannot preserve register-relative labels or numeric local labels.

### Example

```
label   ADC     r2,r3,r4
        KEEP    label        ; makes label available to debuggers
        ADD     r2,r2,r5
```

***Related concepts***

*6.10 Numeric local labels* on page 6-177

***Related reference***

*7.52 MAP* on page 7-259

## 7.49 LCLA, LCLL, and LCLS

The `LCLA`, `LCLL`, and `LCLS` directives declare and initialize local variables.

### Syntax

*`lclx variable`*

where:

*`lclx`*

is one of `LCLA`, `LCLL`, or `LCLS`.

*`variable`*

is the name of the variable. *`variable`* must be unique within the macro that contains it.

### Usage

The `LCLA` directive declares a local arithmetic variable, and initializes its value to 0.

The `LCLL` directive declares a local logical variable, and initializes its value to `{FALSE}`.

The `LCLS` directive declares a local string variable, and initializes its value to a null string, `""`.

Using one of these directives for a variable that is already defined re-initializes the variable.

The scope of the variable is limited to a particular instantiation of the macro that contains it.

Set the value of the variable with a `SETA`, `SETL`, or `SETS` directive.

### Example

```
        MACRO                           ; Declare a macro
$label  message $a                      ; Macro prototype line
        LCLS    err                     ; Declare local string
                                        ; variable err.
err     SETS    "error no: "            ; Set value of err
$label  ; code
        INFO    0, "err":CC::STR:$a     ; Use string
        MEND
```

***Related reference***

*7.42 GBLA, GBLL, and GBLS* on page 7-245

*7.63 SETA, SETL, and SETS* on page 7-272

*7.51 MACRO and MEND* on page 7-256

## 7.50 LTORG

The `LTORG` directive instructs the assembler to assemble the current literal pool immediately.

### Syntax

`LTORG`

### Usage

The assembler assembles the current literal pool at the end of every code section. The end of a code section is determined by the `AREA` directive at the beginning of the following section, or the end of the assembly.

These default literal pools can sometimes be out of range of some `LDR`, `VLDR`, and `WLDR` pseudo-instructions. Use `LTORG` to ensure that a literal pool is assembled within range.

Large programs can require several literal pools. Place `LTORG` directives after unconditional branches or subroutine return instructions so that the processor does not attempt to execute the constants as instructions.

The assembler word-aligns data in literal pools.

### Example

```
        AREA    Example, CODE, READONLY
start   BL      func1
func1                           ; function body
        ; code
        LDR     r1,=0x55555555  ; => LDR R1, [pc, #offset to Literal Pool 1]
        ; code
        MOV     pc,lr           ; end function
        LTORG                   ; Literal Pool 1 contains literal &55555555.
data    SPACE   4200            ; Clears 4200 bytes of memory starting at current location.
        END                     ; Default literal pool is empty.
```

## 7.51 MACRO and MEND

The `MACRO` directive marks the start of the definition of a macro. Macro expansion terminates at the `MEND` directive.

### Syntax

These two directives define a macro. The syntax is:

```
     MACRO
{$label}  macroname{$cond} {$parameter{,$parameter}...}
     ; code
     MEND
```

where:

*`$label`*

is a parameter that is substituted with a symbol given when the macro is invoked. The symbol is usually a label.

*`macroname`*

is the name of the macro. It must not begin with an instruction or directive name.

*`$cond`*

is a special parameter designed to contain a condition code. Values other than valid condition codes are permitted.

*`$parameter`*

is a parameter that is substituted when the macro is invoked. A default value for a parameter can be set using this format:

```
$parameter="default value"
```

Double quotes must be used if there are any spaces within, or at either end of, the default value.

### Usage

If you start any `WHILE...WEND` loops or `IF...ENDIF` conditions within a macro, they must be closed before the `MEND` directive is reached. You can use `MEXIT` to enable an early exit from a macro, for example, from within a loop.

Within the macro body, parameters such as *`$label`*, *`$parameter`* or *`$cond`* can be used in the same way as other variables. They are given new values each time the macro is invoked. Parameters must begin with `$` to distinguish them from ordinary symbols. Any number of parameters can be used.

*`$label`* is optional. It is useful if the macro defines internal labels. It is treated as a parameter to the macro. It does not necessarily represent the first instruction in the macro expansion. The macro defines the locations of any labels.

Use `|` as the argument to use the default value of a parameter. An empty string is used if the argument is omitted.

In a macro that uses several internal labels, it is useful to define each internal label as the base label with a different suffix.

Use a dot between a parameter and following text, or a following parameter, if a space is not required in the expansion. Do not use a dot between preceding text and a parameter.

You can use the *`$cond`* parameter for condition codes. Use the unary operator `:REVERSE_CC:` to find the inverse condition code, and `:CC_ENCODING:` to find the 4-bit encoding of the condition code.

Macros define the scope of local variables.

Macros can be nested.

## Examples

A macro that uses internal labels to implement loops:

```
 ; macro definition
                MACRO                   ; start macro definition
$label          xmac    $p1,$p2
                ; code
$label.loop1    ; code
                ; code
                BGE     $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT     $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND                    ; end macro definition
 ; macro invocation
abc             xmac    subr1,de        ; invoke macro
                ; code                  ; this is what is
abcloop1        ; code                  ; is produced when
                ; code                  ; the xmac macro is
                BGE     abcloop1        ; expanded
abcloop2        ; code
                BL      subr1
                BGT     abcloop2
                ; code
                ADR     de
                ; code
```

A macro that produces assembly-time diagnostics:

```
        MACRO                       ; Macro definition
        diagnose  $param1="default"  ; This macro produces
        INFO      0,"$param1"       ; assembly-time diagnostics
        MEND                        ; (on second assembly pass)
 ; macro expansion
        diagnose           ; Prints blank line at assembly-time
        diagnose "hello"   ; Prints "hello" at assembly-time
        diagnose |         ; Prints "default" at assembly-time
```

When variables are being passed in as arguments, use of | might leave some variables unsubstituted. To work around this, define the | in a `LCLS` or `GBLS` variable and pass this variable as an argument instead of |. For example:

```
        MACRO               ; Macro definition
        m2 $a,$b=r1,$c      ; The default value for $b is r1
        add $a,$b,$c        ; The macro adds $b and $c and puts result in $a.
        MEND                ; Macro end
        MACRO               ; Macro definition
        m1 $a,$b            ; This macro adds $b to r1 and puts result in $a.
        LCLS def            ; Declare a local string variable for |
def     SETS "|"            ; Define |
        m2 $a,$def,$b       ; Invoke macro m2 with $def instead of |
                            ; to use the default value for the second argument.
        MEND                ; Macro end
```

A macro that uses a condition code parameter:

```
        AREA    codx, CODE, READONLY
; macro definition
        MACRO
        Return$cond
        [ {ARCHITECTURE} <> "4"
          BX$cond lr
          |
          MOV$cond pc,lr
        ]
        MEND
; macro invocation
fun     PROC
        CMP      r0,#0
        MOVEQ    r0,#1
        ReturnEQ
        MOV      r0,#0
        Return
        ENDP
        END
```

***Related concepts***

*3.22 Use of macros* on page 3-63

*6.4 Assembly time substitution of variables* on page 6-171

***Related reference***

*7.53 MEXIT* on page 7-260

*7.42 GBLA, GBLL, and GBLS* on page 7-245

*7.49 LCLA, LCLL, and LCLS* on page 7-254

## 7.52 MAP

The `MAP` directive sets the origin of a storage map to a specified address.

### Syntax

`MAP` *`expr{,base-register}`*

where:

*`expr`*

is a numeric or PC-relative expression:

- If *`base-register`* is not specified, *`expr`* evaluates to the address where the storage map starts. The storage map location counter is set to this address.
- If *`expr`* is PC-relative, you must have defined the label before you use it in the map. The map requires the definition of the label during the first pass of the assembler.

*`base-register`*

specifies a register. If *`base-register`* is specified, the address where the storage map starts is the sum of *`expr`*, and the value in *`base-register`* at runtime.

### Usage

Use the `MAP` directive in combination with the `FIELD` directive to describe a storage map.

Specify *`base-register`* to define register-relative labels. The base register becomes implicit in all labels defined by following `FIELD` directives, until the next `MAP` directive. The register-relative labels can be used in load and store instructions.

The `MAP` directive can be used any number of times to define multiple storage maps.

The storage-map location counter, `{VAR}`, is set to the same address as that specified by the `MAP` directive. The `{VAR}` counter is set to zero before the first `MAP` directive is used.

^ is a synonym for `MAP`.

### Examples

```
        MAP     0,r9
        MAP     0xff,r9
```

***Related concepts***

*1.3 How the assembler works* on page 1-19

***Related reference***

*7.29 FIELD* on page 7-232

*1.4 Directives that can be omitted in pass 2 of the assembler* on page 1-21

## 7.53 MEXIT

The MEXIT directive exits a macro definition before the end.

### Usage

Use MEXIT when you require an exit from within the body of a macro. Any unclosed WHILE...WEND loops or IF...ENDIF conditions within the body of the macro are closed by the assembler before the macro is exited.

### Example

```
        MACRO
$abc    example abc     $param1,$param2
        ; code
        WHILE condition1
            ; code
            IF condition2
                ; code
                MEXIT
            ELSE
                ; code
            ENDIF
        WEND
        ; code
        MEND
```

***Related reference***

*7.51 MACRO and MEND* on page 7-256

## 7.54 NOFP

The `NOFP` directive ensures that there are no floating-point instructions in an assembly language source file.

### Syntax

```
NOFP
```

### Usage

Use `NOFP` to ensure that no floating-point instructions are used in situations where there is no support for floating-point instructions either in software or in target hardware.

If a floating-point instruction occurs after the `NOFP` directive, an `Unknown opcode` error is generated and the assembly fails.

If a `NOFP` directive occurs after a floating-point instruction, the assembler generates the error:

```
Too late to ban floating point instructions
```

and the assembly fails.

## 7.55 OPT

The `OPT` directive sets listing options from within the source code.

### Syntax

`OPT` *n*

where:

*n*

is the `OPT` directive setting. The following table lists the valid settings:

**Table 7-2 OPT directive settings**

| OPT n | Effect |
|---|---|
| 1 | Turns on normal listing. |
| 2 | Turns off normal listing. |
| 4 | Page throw. Issues an immediate form feed and starts a new page. |
| 8 | Resets the line number counter to zero. |
| 16 | Turns on listing for `SET`, `GBL` and `LCL` directives. |
| 32 | Turns off listing for `SET`, `GBL` and `LCL` directives. |
| 64 | Turns on listing of macro expansions. |
| 128 | Turns off listing of macro expansions. |
| 256 | Turns on listing of macro invocations. |
| 512 | Turns off listing of macro invocations. |
| 1024 | Turns on the first pass listing. |
| 2048 | Turns off the first pass listing. |
| 4096 | Turns on listing of conditional directives. |
| 8192 | Turns off listing of conditional directives. |
| 16384 | Turns on listing of `MEND` directives. |
| 32768 | Turns off listing of `MEND` directives. |

### Usage

Specify the `--list=` assembler option to turn on listing.

By default the `--list=` option produces a normal listing that includes variable declarations, macro expansions, call-conditioned directives, and `MEND` directives. The listing is produced on the second pass only. Use the `OPT` directive to modify the default listing options from within your code.

You can use `OPT` to format code listings. For example, you can specify a new page before functions and sections.

### Example

```
        AREA    Example, CODE, READONLY
start   ; code
        ; code
        BL      func1
        ; code
        OPT 4                 ; places a page break before func1
func1   ; code
```

***Related reference***

*5.40 --list=file* on page 5-138

## 7.56 QN, DN, and SN

The `QN`, `DN`, and `SN` directives define names for Advanced SIMD and floating-point registers.

### Syntax

`name directive expr{.type}{[x]}`

where:

*`directive`*

is `QN`, `DN`, or `SN`.

*`name`*

is the name to be assigned to the extension register. *`name`* cannot be the same as any of the predefined names.

*`expr`*

Can be:

- An expression that evaluates to a number in the range:
  - — 0-15 if you are using QN in A32/T32 Advanced SIMD code.
  - — 0-31 otherwise.
- A predefined register name, or a register name that has already been defined in a previous directive.

*`type`*

is any Advanced SIMD or floating-point datatype.

`[x]`

is only available for Advanced SIMD code. `[x]` is a scalar index into a register.

*`type`* and `[x]` are *Extended notation*.

### Usage

Use `QN`, `DN`, or `SN` to allocate convenient names to extension registers, to help you to remember what you use each one for.

The `QN` directive defines a name for a specified 128-bit extension register.

The `DN` directive defines a name for a specified 64-bit extension register.

The `SN` directive defines a name for a specified single-precision floating-point register.

**Note**

Avoid conflicting uses of the same register under different names.

You cannot specify a vector length in a `DN` or `SN` directive.

### Examples

```
energy  DN  6   ; defines energy as a symbol for
                ; floating-point double-precision register 6
mass    SN  16  ; defines mass as a symbol for
                ; floating-point single-precision register 16
```

### Extended notation examples

```
varA    DN      d1.U16
varB    DN      d2.U16
varC    DN      d3.U16
        VADD    varA,varB,varC       ; VADD.U16 d1,d2,d3
index   DN      d4.U16[0]
```

```
result  QN      q5.I32
        VMULL   result,varA,index    ; VMULL.U16 q5,d1,d4[0]
```

## 7.57 RELOC

The `RELOC` directive explicitly encodes an ELF relocation in an object file.

### Syntax

`RELOC` *`n`*`,` *`symbol`*

`RELOC` *`n`*

where:

*`n`*

must be an integer in the range 0 to 255 or one of the relocation names defined in the *Application Binary Interface for the Arm® Architecture*.

*`symbol`*

can be any PC-relative label.

### Usage

Use `RELOC` *`n`*, *`symbol`* to create a relocation with respect to the address labeled by *`symbol`*.

If used immediately after an A32 or T32 instruction, `RELOC` results in a relocation at that instruction. If used immediately after a `DCB`, `DCW`, or `DCD`, or any other data generating directive, `RELOC` results in a relocation at the start of the data. Any addend to be applied must be encoded in the instruction or in the data.

If the assembler has already emitted a relocation at that place, the relocation is updated with the details in the `RELOC` directive, for example:

```
DCD     sym2 ; R_ARM_ABS32 to sym32
RELOC   55   ; ... makes it R_ARM_ABS32_NOI
```

`RELOC` is faulted in all other cases, for example, after any non-data generating directive, `LTORG`, `ALIGN`, or as the first thing in an `AREA`.

Use `RELOC` *`n`* to create a relocation with respect to the anonymous symbol, that is, symbol 0 of the symbol table. If you use `RELOC` *`n`* without a preceding assembler generated relocation, the relocation is with respect to the anonymous symbol.

### Examples

```
IMPORT  impsym
LDR     r0,[pc,#-8]
RELOC   4, impsym
DCD     0
RELOC   2, sym
DCD     0,1,2,3,4       ; the final word is relocated
RELOC   38,sym2         ; R_ARM_TARGET1
DCD     impsym
RELOC   R_ARM_TARGET1   ; relocation code 38
```

***Related information***

*Application Binary Interface for the Arm Architecture*

## 7.58 REQUIRE

The `REQUIRE` directive specifies a dependency between sections.

### Syntax

`REQUIRE` *`label`*

where:

*`label`*

is the name of the required label.

### Usage

Use `REQUIRE` to ensure that a related section is included, even if it is not directly called. If the section containing the `REQUIRE` directive is included in a link, the linker also includes the section containing the definition of the specified label.

## 7.59 REQUIRE8 and PRESERVE8

The `REQUIRE8` and `PRESERVE8` directives specify that the current file requires or preserves eight-byte alignment of the stack.

**Note**

This directive is required to support non-ABI conforming toolchains. It has no effect on AArch64 assembly and is not required when targeting AArch64.

---

### Syntax

`REQUIRE8 {`*`bool`*`}`

`PRESERVE8 {`*`bool`*`}`

where:

*`bool`*

is an optional Boolean constant, either `{TRUE}` or `{FALSE}`.

### Usage

Where required, if your code preserves eight-byte alignment of the stack, use `PRESERVE8` to set the `PRES8` build attribute on your file. If your code does not preserve eight-byte alignment of the stack, use `PRESERVE8 {FALSE}` to ensure that the `PRES8` build attribute is not set. Use `REQUIRE8` to set the `REQ8` build attribute. If there are multiple `REQUIRE8` or `PRESERVE8` directives in a file, the assembler uses the value of the last directive.

The linker checks that any code that requires eight-byte alignment of the stack is only called, directly or indirectly, by code that preserves eight-byte alignment of the stack.

**Note**

If you omit both `PRESERVE8` and `PRESERVE8 {FALSE}`, the assembler decides whether to set the `PRES8` build attribute or not, by examining instructions that modify the SP. Arm recommends that you specify `PRESERVE8` explicitly.

You can enable a warning by using the `--diag_warning 1546` option when invoking `armasm`.

This gives you warnings like:

```
"test.s", line 37: Warning: A1546W: Stack pointer update potentially breaks 8 byte stack
alignment
       37 00000044         STMFD    sp!,{r2,r3,lr}
```

---

### Examples

```
REQUIRE8
REQUIRE8     {TRUE}      ; equivalent to REQUIRE8
REQUIRE8     {FALSE}     ; equivalent to absence of REQUIRE8
PRESERVE8    {TRUE}      ; equivalent to PRESERVE8
PRESERVE8    {FALSE}     ; NOT exactly equivalent to absence of PRESERVE8
```

***Related reference***

*5.21 --diag_warning=tag[,tag,...]* on page 5-119

***Related information***

*Eight-byte Stack Alignment*

## 7.60 RLIST

The `RLIST` (register list) directive gives a name to a set of general-purpose registers in A32/T32 code.

### Syntax

*`name`* `RLIST {`*`list-of-registers`*`}`

where:

*`name`*
: is the name to be given to the set of registers. ***`name`*** cannot be the same as any of the predefined names.

*`list-of-registers`*
: is a comma-delimited list of register names and register ranges. The register list must be enclosed in braces.

### Usage

Use `RLIST` to give a name to a set of registers to be transferred by the `LDM` or `STM` instructions.

`LDM` and `STM` always put the lowest physical register numbers at the lowest address in memory, regardless of the order they are supplied to the `LDM` or `STM` instruction. If you have defined your own symbolic register names it can be less apparent that a register list is not in increasing register order.

Use the `--diag_warning 1206` assembler option to ensure that the registers in a register list are supplied in increasing register order. If registers are not supplied in increasing register order, a warning is issued.

### Example

```
Context RLIST   {r0-r6,r8,r10-r12,pc}
```

## 7.61 RN

The `RN` directive defines a name for a specified register.

### Syntax

*`name`* `RN` *`expr`*

where:

*`name`*

is the name to be assigned to the register. *`name`* cannot be the same as any of the predefined names.

*`expr`*

evaluates to a register number from 0 to 15.

### Usage

Use `RN` to allocate convenient names to registers, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

### Examples

```
regname     RN  11  ; defines regname for register 11
sqr4        RN  r6  ; defines sqr4 for register 6
```

## 7.62 ROUT

The `ROUT` directive marks the boundaries of the scope of numeric local labels.

### Syntax

`{name} ROUT`

where:

*`name`*

is the name to be assigned to the scope.

### Usage

Use the `ROUT` directive to limit the scope of numeric local labels. This makes it easier for you to avoid referring to a wrong label by accident. The scope of numeric local labels is the whole area if there are no `ROUT` directives in it.

Use the *`name`* option to ensure that each reference is to the correct numeric local label. If the name of a label or a reference to a label does not match the preceding `ROUT` directive, the assembler generates an error message and the assembly fails.

### Example

```
            ; code
routineA    ROUT            ; ROUT is not necessarily a routine
            ; code
3routineA   ; code          ; this label is checked
            ; code
            BEQ     %4routineA   ; this reference is checked
            ; code
            BGE     %3      ; refers to 3 above, but not checked
            ; code
4routineA   ; code          ; this label is checked
            ; code
otherstuff  ROUT            ; start of next scope
```

***Related concepts***

*6.10 Numeric local labels* on page 6-177

***Related reference***

*7.6 AREA* on page 7-205

## 7.63 SETA, SETL, and SETS

The `SETA`, `SETL`, and `SETS` directives set the value of a local or global variable.

### Syntax

*`variable setx expr`*

where:

*`variable`*

is the name of a variable declared by a `GBLA`, `GBLL`, `GBLS`, `LCLA`, `LCLL`, or `LCLS` directive.

*`setx`*

is one of `SETA`, `SETL`, or `SETS`.

*`expr`*

is an expression that is:
- Numeric, for `SETA`.
- Logical, for `SETL`.
- String, for `SETS`.

### Usage

The `SETA` directive sets the value of a local or global arithmetic variable.

The `SETL` directive sets the value of a local or global logical variable.

The `SETS` directive sets the value of a local or global string variable.

You must declare *`variable`* using a global or local declaration directive before using one of these directives.

You can also predefine variable names on the command line.

### Restrictions

The value you can specify using a `SETA` directive is limited to 32 bits. If you exceed this limit, the assembler reports an error. A possible workaround in A64 code is to use an `EQU` directive instead of `SETA`, although `EQU` defines a constant, whereas `GBLA` and `SETA` define a variable.

For example, replace the following code:

```
                    GBLA    MyAddress
    MyAddress       SETA    0x0000008000000000
```

with:

```
    MyAddress       EQU     0x0000008000000000
```

### Examples

```
                GBLA    VersionNumber
VersionNumber   SETA    21
                GBLL    Debug
Debug           SETL    {TRUE}
                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```

***Related concepts***

*6.12 String expressions* on page 6-179

*6.14 Numeric expressions* on page 6-181

*6.17 Logical expressions* on page 6-184

***Related reference***

*7.42 GBLA, GBLL, and GBLS* on page 7-245

*7.49 LCLA, LCLL, and LCLS* on page 7-254

*5.54 --predefine "directive"* on page 5-152

## 7.64 SPACE or FILL

The `SPACE` directive reserves a zeroed block of memory. The `FILL` directive reserves a block of memory to fill with a given value.

### Syntax

`{`*`label`*`} SPACE `*`expr`*

`{`*`label`*`} FILL `*`expr`*`{,`*`value`*`{,`*`valuesize`*`}}`

where:

*`label`*

is an optional label.

*`expr`*

evaluates to the number of bytes to fill or zero.

*`value`*

evaluates to the value to fill the reserved bytes with. *`value`* is optional and if omitted, it is 0. *`value`* must be 0 in a `NOINIT` area.

*`valuesize`*

is the size, in bytes, of *`value`*. It can be any of 1, 2, or 4. *`valuesize`* is optional and if omitted, it is 1.

### Usage

Use the `ALIGN` directive to align any code following a `SPACE` or `FILL` directive.

`%` is a synonym for `SPACE`.

### Example

```
        AREA    MyData, DATA, READWRITE
data1   SPACE   255       ; defines 255 bytes of zeroed store
data2   FILL    50,0xAB,1 ; defines 50 bytes containing 0xAB
```

***Related concepts***

*6.14 Numeric expressions* on page 6-181

***Related reference***

*7.5 ALIGN* on page 7-203

*7.15 DCB* on page 7-217

*7.16 DCD and DCDU* on page 7-218

*7.21 DCQ and DCQU* on page 7-223

*7.22 DCW and DCWU* on page 7-224

## 7.65 THUMB directive

The `THUMB` directive instructs the assembler to interpret subsequent instructions as T32 instructions, using the UAL syntax.

——— **Note** ———

Not supported for AArch64 state.

———————————

### Syntax

`THUMB`

### Usage

In files that contain code using different instruction sets, the `THUMB` directive must precede T32 code written in UAL syntax.

If necessary, this directive also inserts one byte of padding to align to the next halfword boundary.

This directive does not assemble to any instructions. It also does not change the state. It only instructs `armasm` to assemble T32 instructions as appropriate, and inserts padding if necessary.

### Example

This example shows how you can use `ARM` and `THUMB` directives to switch state and assemble both A32 and T32 instructions in a single area.

```
        AREA ToT32, CODE, READONLY      ; Name this block of code
        ENTRY                           ; Mark first instruction to execute
        ARM                             ; Subsequent instructions are A32
start
        ADR     r0, into_t32 + 1        ; Processor starts in A32 state
        BX      r0                      ; Inline switch to T32 state
        THUMB                           ; Subsequent instructions are T32
into_t32
        MOVS    r0, #10                 ; New-style T32 instructions
```

***Related reference***

*7.7 ARM or CODE32 directive* on page 7-209

*7.11 CODE16 directive* on page 7-213

## 7.66 TTL and SUBT

The `TTL` directive inserts a title at the start of each page of a listing file. The `SUBT` directive places a subtitle on the pages of a listing file.

### Syntax

`TTL` *`title`*

`SUBT` *`subtitle`*

where:

*`title`*

is the title.

*`subtitle`*

is the subtitle.

### Usage

Use the `TTL` directive to place a title at the top of each page of a listing file. If you want the title to appear on the first page, the `TTL` directive must be on the first line of the source file.

Use additional `TTL` directives to change the title. Each new `TTL` directive takes effect from the top of the next page.

Use `SUBT` to place a subtitle at the top of each page of a listing file. Subtitles appear in the line below the titles. If you want the subtitle to appear on the first page, the `SUBT` directive must be on the first line of the source file.

Use additional `SUBT` directives to change subtitles. Each new `SUBT` directive takes effect from the top of the next page.

### Examples

```
    TTL   First Title    ; places title on first and subsequent pages of listing file.
    SUBT  First Subtitle ; places subtitle on second and subsequent pages of listing file.
```

## 7.67 WHILE and WEND

The `WHILE` directive starts a sequence of instructions or directives that are to be assembled repeatedly. The sequence is terminated with a `WEND` directive.

### Syntax

```
WHILE logical-expression
  code
WEND
```

where:

*`logical-expression`*

is an expression that can evaluate to either `{TRUE}` or `{FALSE}`.

### Usage

Use the `WHILE` directive, together with the `WEND` directive, to assemble a sequence of instructions a number of times. The number of repetitions can be zero.

You can use `IF...ENDIF` conditions within `WHILE...WEND` loops.

`WHILE...WEND` loops can be nested.

### Example

```
        GBLA count                  ; declare local variable
count   SETA    1                   ; you are not restricted to
        WHILE   count <= 4          ; such simple conditions
count   SETA    count+1             ; In this case, this code is
            ; code                  ; executed four times
            ; code                  ;
        WEND
```

***Related concepts***

*6.17 Logical expressions* on page 6-184

***Related reference***

*7.2 About assembly control directives* on page 7-200

## 7.68 WN and XN

The `WN`, and `XN` directives define names for registers in A64 code.

The `WN` directive defines a name for a specified 32-bit register.

The `XN` directive defines a name for a specified 64-bit register.

### Syntax

*`name directive expr`*

where:

*`name`*
: is the name to be assigned to the register. *`name`* cannot be the same as any of the predefined names.

*`directive`*
: is `WN` or `XN`.

*`expr`*
: evaluates to a register number from 0 to 30.

### Usage

Use `WN` and `XN` to allocate convenient names to registers in A64 code, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

### Examples

```
sqr4        WN w16  ; defines sqr4 for register w16
regname     XN 21  ; defines regname for register x21
```

# Chapter 8
# armasm-Specific Instruction Set Features

Describes the additional support that `armasm` provides for the Arm instruction set.

It contains the following sections:

## 8.1 armasm support for the CSDB instruction

For conditional CSDB instructions that specify a condition `{c}` other than `AL` in A32, and for any condition `{c}` used inside an `IT` block in T32, then `armasm` rejects conditional CSDB instructions, outputs an error message, and aborts.

For example:

- For A32 code:

```
"test2.s", line 4: Error: A1895E: The specified condition results in UNPREDICTABLE
behaviour
                    4 00000000   CSDBEQ
```

- For T32 code:

```
"test2.s", line 8: Error: A1603E: This instruction inside IT block has UNPREDICTABLE
results
                    8 00000006   CSDBEQ
```

You can relax this behavior by using:
- The `--diag-suppress=1895` option for A32 code.
- The `--diag-suppress=1603` option for T32 code.

You can also use the `--unsafe` option with these options. However, this option disables many correctness checks.

## 8.2 A32 and T32 pseudo-instruction summary

An overview of the pseudo-instructions available in the A32 and T32 instruction sets.

**Table 8-1 Summary of pseudo-instructions**

| Mnemonic | Brief description | See |
|---|---|---|
| `ADRL` pseudo-instruction | Load program or register-relative address (medium range) | *8.3 ADRL pseudo-instruction* on page 8-282 |
| `CPY` pseudo-instruction | Copy | *8.4 CPY pseudo-instruction* on page 8-284 |
| `LDR` pseudo-instruction | Load Register pseudo-instruction | *8.5 LDR pseudo-instruction* on page 8-285 |
| `MOV32` pseudo-instruction | Move 32-bit immediate to register | *8.6 MOV32 pseudo-instruction* on page 8-287 |
| `NEG` pseudo-instruction | Negate | *8.7 NEG pseudo-instruction* on page 8-288 |
| `UND` pseudo-instruction | Generate an architecturally undefined instruction. | *8.8 UND pseudo-instruction* on page 8-289 |

## 8.3 ADRL pseudo-instruction

Load a PC-relative or register-relative address into a register.

### Syntax

`ADRL{cond} Rd,label`

where:

*`cond`*
: is an optional condition code.

*`Rd`*
: is the register to load.

*`label`*
: is a PC-relative or register-relative expression.

### Usage

`ADRL` always assembles to two 32-bit 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. You can use the `LDR` pseudo-instruction for loading a wider range of addresses.

`ADRL` is similar to the `ADR` instruction, except `ADRL` can load a wider range of addresses because it generates two data processing instructions.

`ADRL` produces position-independent code, because the address is PC-relative or register-relative.

If *`label`* is PC-relative, it must evaluate to an address in the same assembler area as the `ADRL` pseudo-instruction.

If you use `ADRL` to generate a target for a `BX` or `BLX` instruction, it is your responsibility to set the T32 bit (bit 0) of the address if the target contains T32 instructions.

### Architectures and range

The available range depends on the instruction set in use:

**A32**
: The range of the instruction is any value that can be generated by two `ADD` or two `SUB` instructions. That is, any value that can be produced by the addition of two values, each of which is 8 bits rotated right by any even number of bits within a 32-bit word.

**T32, 32-bit encoding**
: ±1MB bytes to a byte, halfword, or word-aligned address.

**T32, 16-bit encoding**
: `ADRL` is not available.

The given range is relative to a point four bytes (in T32 code) or two words (in A32 code) after the address of the current instruction.

**Note**

`ADRL` is not available in Armv6-M and Armv8-M Baseline.

***Related concepts***

*6.5 Register-relative and PC-relative expressions* on page 6-172

*3.4 Load immediate values* on page 3-39

***Related reference***

*8.5 LDR pseudo-instruction* on page 8-285

***Related information***

*Arm Architecture Reference Manual*

## 8.4 CPY pseudo-instruction

Copy a value from one register to another.

### Syntax

`CPY{cond} Rd, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm*
: is the register holding the value to be copied.

### Operation

The `CPY` pseudo-instruction copies a value from one register to another, without changing the condition flags.

`CPY Rd, Rm` assembles to `MOV Rd, Rm`.

### Architectures

This pseudo-instruction is available in A32 code and in T32 code.

### Register restrictions

Using SP or PC for both *Rd* and *Rm* is deprecated.

### Condition flags

This instruction does not change the condition flags.

***Related reference***
*MOV (A32/T32)*

## 8.5 LDR pseudo-instruction

Load a register with either a 32-bit immediate value or an address.

**Note**

This describes the `LDR` pseudo-instruction only, and not the `LDR` instruction.

### Syntax

`LDR{cond}{.W} Rt, =expr`

`LDR{cond}{.W} Rt, =label_expr`

where:

*cond*

is an optional condition code.

`.W`

is an optional instruction width specifier.

*Rt*

is the register to be loaded.

*expr*

evaluates to a numeric value.

*label_expr*

is a PC-relative or external expression of an address in the form of a label plus or minus a numeric value.

### Usage

When using the `LDR` pseudo-instruction:

- If the value of *expr* can be loaded with a valid `MOV` or `MVN` instruction, the assembler uses that instruction.
- If a valid `MOV` or `MVN` instruction cannot be used, or if the `label_expr` syntax is used, the assembler places the constant in a literal pool and generates a PC-relative `LDR` instruction that reads the constant from the literal pool.

  **Note**

  — An address loaded in this way is fixed at link time, so the code is not position-independent.
  — The address holding the constant remains valid regardless of where the linker places the ELF section containing the `LDR` instruction.

The assembler places the value of *label_expr* in a literal pool and generates a PC-relative `LDR` instruction that loads the value from the literal pool.

If *label_expr* is an external expression, or is not contained in the current section, the assembler places a linker relocation directive in the object file. The linker generates the address at link time.

If *label_expr* is either a named or numeric local label, the assembler places a linker relocation directive in the object file and generates a symbol for that local label. The address is generated at link time. If the local label references T32 code, the T32 bit (bit 0) of the address is set.

The offset from the PC to the value in the literal pool must be less than ±4KB (in an A32 or 32-bit T32 encoding) or in the range 0 to +1KB (16-bit T32 encoding). You are responsible for ensuring that there is a literal pool within range.

If the label referenced is in T32 code, the `LDR` pseudo-instruction sets the T32 bit (bit 0) of *`label_expr`*.

**Note**

In *RealView® Compilation Tools* (RVCT) v2.2, the T32 bit of the address was not set. If you have code that relies on this behavior, use the command line option `--untyped_local_labels` to force the assembler not to set the T32 bit when referencing labels in T32 code.

### LDR in T32 code

You can use the `.W` width specifier to force `LDR` to generate a 32-bit instruction in T32 code. `LDR.W` always generates a 32-bit instruction, even if the immediate value could be loaded in a 16-bit `MOV`, or there is a literal pool within reach of a 16-bit PC-relative load.

If the value to be loaded is not known in the first pass of the assembler, `LDR` without `.W` generates a 16-bit instruction in T32 code, even if that results in a 16-bit PC-relative load for a value that could be generated in a 32-bit `MOV` or `MVN` instruction. However, if the value is known in the first pass, and it can be generated using a 32-bit `MOV` or `MVN` instruction, the `MOV` or `MVN` instruction is used.

In UAL syntax, the `LDR` pseudo-instruction never generates a 16-bit flag-setting `MOV` instruction. Use the `--diag_warning 1727` assembler command line option to check when a 16-bit instruction could have been used.

You can use the `MOV32` pseudo-instruction for generating immediate values or addresses without loading from a literal pool.

### Examples

```
        LDR     r3,=0xff0     ; loads 0xff0 into R3
                              ; =>  MOV.W r3,#0xff0
        LDR     r1,=0xfff     ; loads 0xfff into R1
                              ; =>  LDR r1,[pc,offset_to_litpool]
                              ;     ...
                              ;     litpool DCD 0xfff
        LDR     r2,=place     ; loads the address of
                              ; place into R2
                              ; =>  LDR r2,[pc,offset_to_litpool]
                              ;     ...
                              ;     litpool DCD place
```

***Related concepts***

*6.3 Numeric constants* on page 6-170

*6.5 Register-relative and PC-relative expressions* on page 6-172

*6.10 Numeric local labels* on page 6-177

***Related reference***

*5.62 --untyped_local_labels* on page 5-160

*8.6 MOV32 pseudo-instruction* on page 8-287

## 8.6 MOV32 pseudo-instruction

Load a register with either a 32-bit immediate value or any address.

### Syntax

`MOV32{cond} Rd, expr`

where:

*`cond`*
: is an optional condition code.

*`Rd`*
: is the register to be loaded. *`Rd`* must not be SP or PC.

*`expr`*
: can be any one of the following:

  *`symbol`*
  : A label in this or another program area.

  *`#constant`*
  : Any 32-bit immediate value.

  *`symbol + constant`*
  : A label plus a 32-bit immediate value.

### Usage

`MOV32` always generates two 32-bit instructions, a `MOV`, `MOVT` pair. This enables you to load any 32-bit immediate, or to access the whole 32-bit address space.

The main purposes of the `MOV32` pseudo-instruction are:

- To generate literal constants when an immediate value cannot be generated in a single instruction.
- To load a PC-relative or external address into a register. The address remains valid regardless of where the linker places the ELF section containing the `MOV32`.

  **Note**

  An address loaded in this way is fixed at link time, so the code is not position-independent.

`MOV32` sets the T32 bit (bit 0) of the address if the label referenced is in T32 code.

### Architectures

This pseudo-instruction is available in A32 and T32.

### Examples

```
    MOV32 r3, #0xABCDEF12  ; loads 0xABCDEF12 into R3
    MOV32 r1, Trigger+12   ; loads the address that is 12 bytes
                           ; higher than the address Trigger into R1
```

***Related reference***

*Condition code suffixes*

## 8.7 NEG pseudo-instruction

Negate the value in a register.

### Syntax

`NEG{cond} Rd, Rm`

where:

*`cond`*
is an optional condition code.

*`Rd`*
is the destination register.

*`Rm`*
is the register containing the value that is subtracted from zero.

### Operation

The `NEG` pseudo-instruction negates the value in one register and stores the result in a second register.

`NEG{cond} Rd, Rm` assembles to `RSBS{cond} Rd, Rm, #0`.

### Architectures

The 32-bit encoding of this pseudo-instruction is available in A32 and T32.

There is no 16-bit encoding of this pseudo-instruction available T32.

### Register restrictions

In A32 instructions, using SP or PC for *`Rd`* or *`Rm`* is deprecated. In T32 instructions, you cannot use SP or PC for *`Rd`* or *`Rm`*.

### Condition flags

This pseudo-instruction updates the condition flags, based on the result.

***Related reference***
*ADD (A32/T32)*

## 8.8 UND pseudo-instruction

Generate an architecturally undefined instruction.

### Syntax

`UND{`*`cond`*`}{.W} {#`*`expr`*`}`

where:

*`cond`*
is an optional condition code.

`.W`
is an optional instruction width specifier.

*`expr`*
evaluates to a numeric value. The following table shows the range and encoding of *`expr`* in the instruction, where Y shows the locations of the bits that encode for *`expr`* and V is the 4 bits that encode for the condition code.

If *`expr`* is omitted, the value 0 is used.

**Table 8-2 Range and encoding of expr**

| Instruction | Encoding | Number of bits for *expr* | Range |
|---|---|---|---|
| A32 | 0xV7FYYYFY | 16 | 0-65535 |
| T32 32-bit encoding | 0xF7FYAYFY | 12 | 0-4095 |
| T32 16-bit encoding | 0xDEYY | 8 | 0-255 |

### Usage

An attempt to execute an undefined instruction causes the Undefined instruction exception. Architecturally undefined instructions are expected to remain undefined.

### UND in T32 code

You can use the `.W` width specifier to force `UND` to generate a 32-bit instruction in T32 code. `UND.W` always generates a 32-bit instruction, even if *`expr`* is in the range 0-255.

### Disassembly

The encodings that this pseudo-instruction produces disassemble to `DCI`.

***Related reference***
*Condition code suffixes*

# Chapter 9
# Via File Syntax

Describes the syntax of via files accepted by `armasm`.

It contains the following sections:

## 9.1 Overview of via files

Via files are plain text files that allow you to specify assembler command-line arguments and options.

Typically, you use a via file to overcome the command-line length limitations. However, you might want to create multiple via files that:

- Group similar arguments and options together.
- Contain different sets of arguments and options to be used in different scenarios.

**Note**

In general, you can use a via file to specify any command-line option to a tool, including `--via`. This means that you can call multiple nested via files from within a via file.

### Via file evaluation

When the assembler is invoked it:

1. Replaces the first specified `--via` *`via_file`* argument with the sequence of argument words extracted from the via file, including recursively processing any nested `--via` commands in the via file.
2. Processes any subsequent `--via` *`via_file`* arguments in the same way, in the order they are presented.

That is, via files are processed in the order you specify them, and each via file is processed completely including processing nested via files before processing the next via file.

***Related reference***

*9.2 Via file syntax rules* on page 9-292

*5.64 --via=filename* on page 5-162

## 9.2 Via file syntax rules

Via files must conform to some syntax rules.

- A via file is a text file containing a sequence of words. Each word in the text file is converted into an argument string and passed to the tool.
- Words are separated by whitespace, or the end of a line, except in delimited strings, for example:

  `--bigend --reduce_paths` (two words)

  `--bigend--reduce_paths` (one word)
- The end of a line is treated as whitespace, for example:

  ```
  --bigend
  --reduce_paths
  ```

  This is equivalent to:

  `--bigend --reduce_paths`
- Strings enclosed in quotation marks (`"`), or apostrophes (`'`) are treated as a single word. Within a quoted word, an apostrophe is treated as an ordinary character. Within an apostrophe delimited word, a quotation mark is treated as an ordinary character.

  Use quotation marks to delimit filenames or path names that contain spaces, for example:

  `--errors C:\My Project\errors.txt` (three words)

  `--errors "C:\My Project\errors.txt"` (two words)

  Use apostrophes to delimit words that contain quotes, for example:

  `-DNAME='"ARM Compiler"'` (one word)
- Characters enclosed in parentheses are treated as a single word, for example:

  `--option(x, y, z)` (one word)

  `--option (x, y, z)` (two words)
- Within quoted or apostrophe delimited strings, you can use a backslash (\) character to escape the quote, apostrophe, and backslash characters.
- A word that occurs immediately next to a delimited word is treated as a single word, for example:

  `--errors"C:\Project\errors.txt"`

  This is treated as the single word:

  `--errorsC:\Project\errors.txt`
- Lines beginning with a semicolon (`;`) or a hash (`#`) character as the first nonwhitespace character are comment lines. A semicolon or hash character that appears anywhere else in a line is not treated as the start of a comment, for example:

  ```
  -o objectname.axf      ;this is not a comment
  ```

  A comment ends at the end of a line, or at the end of the file. There are no multi-line comments, and there are no part-line comments.

***Related concepts***

*9.1 Overview of via files* on page 9-291

***Related reference***

*5.64 --via=filename* on page 5-162