

ARM[®] Compiler

Version 6.02

Migration and Compatibility Guide

ARM[®]

ARM® Compiler

Migration and Compatibility Guide

Copyright © 2014, 2015 ARM. All rights reserved.

Release Information

Document History

Issue	Date	Confidentiality	Change
A	14 March 2014	Non-Confidential	ARM v6.00 Release
B	15 December 2014	Non-Confidential	ARM v6.01 Release
C	30 June 2015	Non-Confidential	ARM v6.02 Release

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, ARM makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to ARM’s customers is not intended to create or refer to any partnership relationship with any other company. ARM may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow ARM’s trademark usage guidelines at <http://www.arm.com/about/trademark-usage-guidelines.php>

Copyright © [2014, 2015], ARM Limited or its affiliates. All rights reserved.

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

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 Migration and Compatibility Guide

	Preface	
	<i>About this book</i>	8
Chapter 1	Compiler Configuration Information	
	1.1 <i>Compiler configuration information</i>	1-11
Chapter 2	Command-line Options Comparison	
	2.1 <i>Comparison of ARM Compiler 6 compiler command-line options and older versions of ARM Compiler</i>	2-13
	2.2 <i>Command-line options for preprocessing assembly source code</i>	2-15
Chapter 3	Compiler Source Code Compatibility	
	3.1 <i>Language extension compatibility: keywords</i>	3-17
	3.2 <i>Language extension compatibility: attributes</i>	3-18
	3.3 <i>Language extension compatibility: pragmas</i>	3-19
	3.4 <i>Diagnostics for pragma compatibility</i>	3-22
	3.5 <i>C and C++ implementation compatibility</i>	3-24
	3.6 <i>Compatibility of C++ objects</i>	3-26
Chapter 4	Migrating ARM syntax assembly code to GNU syntax	
	4.1 <i>Overview of differences between ARM and GNU syntax assembly code</i>	4-28
	4.2 <i>Comments</i>	4-30
	4.3 <i>Labels</i>	4-31
	4.4 <i>Numeric local labels</i>	4-32

4.5	<i>Functions</i>	4-34
4.6	<i>Sections</i>	4-35
4.7	<i>Symbol naming rules</i>	4-36
4.8	<i>Numeric literals</i>	4-37
4.9	<i>Operators</i>	4-38
4.10	<i>Alignment</i>	4-39
4.11	<i>PC-relative addressing</i>	4-40
4.12	<i>Conditional directives</i>	4-41
4.13	<i>Data definition directives</i>	4-42
4.14	<i>Instruction set directives</i>	4-43
4.15	<i>Miscellaneous directives</i>	4-44
4.16	<i>Symbol definition directives</i>	4-45

List of Tables

ARM® Compiler Migration and Compatibility Guide

Table 1-1	<i>FlexNet versions</i>	1-11
Table 2-1	<i>Comparison of ARM Compiler 6 compiler command-line options and older versions of ARM Compiler</i>	2-13
Table 3-1	<i>Keyword language extensions that must be replaced</i>	3-17
Table 3-2	<i>Pragma language extensions that must be replaced</i>	3-19
Table 3-3	<i>Pragma diagnostics</i>	3-22
Table 3-4	<i>C and C++ implementation detail differences</i>	3-24
Table 4-1	<i>Operator translation</i>	4-38
Table 4-2	<i>Conditional directive translation</i>	4-41
Table 4-3	<i>Data definition directives translation</i>	4-42
Table 4-4	<i>Instruction set directives translation</i>	4-43
Table 4-5	<i>Miscellaneous directives translation</i>	4-44
Table 4-6	<i>Symbol definition directives translation</i>	4-45

Preface

This preface introduces the *ARM® Compiler Migration and Compatibility Guide*.

It contains the following:

- [About this book on page 8.](#)

About this book

The ARM® Compiler Migration and Compatibility Guide provides migration and compatibility information for users moving from older versions of ARM Compiler to ARM Compiler 6.

Using this book

This book is organized into the following chapters:

Chapter 1 Compiler Configuration Information

Summarizes the locales and FlexNet versions supported by the ARM® compilation tools.

Chapter 2 Command-line Options Comparison

Compares ARM Compiler 6 command-line options to older versions of ARM Compiler.

Chapter 3 Compiler Source Code Compatibility

Provides details of source code compatibility between ARM Compiler 6 and older `armcc` compiler versions.

Chapter 4 Migrating ARM syntax assembly code to GNU syntax

Describes how to migrate assembly code from the legacy ARM syntax (used by `armasm`) to GNU syntax (used by `armclang`).

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.

`monospace`

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.
- The number ARM DUI0742C.
- 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 Information Center](#).
- [ARM Technical Support Knowledge Articles](#).
- [Support and Maintenance](#).
- [ARM Glossary](#).

Chapter 1

Compiler Configuration Information

Summarizes the locales and FlexNet versions supported by the ARM® compilation tools.

It contains the following sections:

- [1.1 Compiler configuration information on page 1-11.](#)

1.1 Compiler configuration information

Summarizes the locales and FlexNet versions supported by the ARM compilation tools.

FlexNet versions in the compilation tools

Different versions of ARM Compiler support different versions of FlexNet.

The FlexNet versions in the compilation tools are:

Table 1-1 FlexNet versions

Compilation tools version	Windows	Linux
ARM Compiler 6.01 and later	11.12.1.0	11.12.1.0
ARM Compiler 6.00	11.10.1.0	11.10.1.0

Locale support in the compilation tools

ARM Compiler only supports the English locale.

Related information

[ARM DS-5 License Management Guide.](#)

Chapter 2

Command-line Options Comparison

Compares ARM Compiler 6 command-line options to older versions of ARM Compiler.

It contains the following sections:

- *2.1 Comparison of ARM Compiler 6 compiler command-line options and older versions of ARM Compiler on page 2-13.*
- *2.2 Command-line options for preprocessing assembly source code on page 2-15.*

2.1 Comparison of ARM Compiler 6 compiler command-line options and older versions of ARM Compiler

ARM Compiler provides many command-line options, including most Clang command-line options as well as a number of ARM-specific options.

The following table describes the most common ARM Compiler command-line options, and shows equivalent options for ARM Compiler 6 and older versions of ARM Compiler.

Additional information about command-line options is available:

- The *armclang Reference Guide* provides more detail about a number of command-line options.
- For a full list of Clang command-line options, consult the Clang and LLVM documentation.

Table 2-1 Comparison of ARM Compiler 6 compiler command-line options and older versions of ARM Compiler

Older ARM Compiler option	ARM Compiler 6 option	Description
-c	-c	Performs the compilation step, but not the link step.
--c90	-xc -std=c90	Enables the compilation of C90 source code. -xc is a positional argument and only affects subsequent input files on the command line
--c99	-xc -std=c99	Enables the compilation of C99 source code. -xc is a positional argument and only affects subsequent input files on the command line
--cpp	-xc++ -std=c++98	Enables the compilation of C++ source code. -xc++ is a positional argument and only affects subsequent input files on the command line
--cpu 8-A.32	--target=arm-arm-none-eabi -march=armv8-a	Targets ARMv8-A, AArch32 state.
--cpu 8-A.64	--target=aarch64-arm-none-eabi	Targets ARMv8-A, AArch64 state. (Implies -march=armv8-a if -mcpu is not specified.)
--cpu 7	--target=arm-arm-none-eabi -march=armv7-a	Targets ARMv7-A.
--cpu=Cortex-A15	--target=arm-arm-none-eabi -mcpu=cortex-a15	Targets the Cortex A15 processor.
-D	-D	Defines a preprocessing macro.
-E	-E	Executes only the preprocessor step.
--fpmode=fast	-ffast-math	Performs aggressive floating-point optimizations at the cost of IEEE compliance.
-I	-I	Adds the specified directories to the list of places that are searched to find included files.
--inline	Default at -O2 and -O3.	There is no equivalent of the --inline option. ARM Compiler 6 automatically decides whether to inline functions at optimization levels -O2 and -O3.
-L	-Xlinker	Specifies command-line options to pass to the linker when a link step is being performed after compilation.

Table 2-1 Comparison of ARM Compiler 6 compiler command-line options and older versions of ARM Compiler (continued)

Older ARM Compiler option	ARM Compiler 6 option	Description
<code>--licretry</code>	No equivalent.	There is no equivalent of the <code>--licretry</code> option. The ARM Compiler 6 tools automatically retry failed attempts to obtain a license.
<code>-M</code>	<code>-M</code>	Instructs the compiler to produce a list of makefile dependency lines suitable for use by a make utility.
<code>-o</code>	<code>-o</code>	Specifies the name of the output file.
<code>-Onum</code>	<code>-Onum</code>	Specifies the level of optimization to be used when compiling source files. The default for older compiler versions is <code>-O2</code> . The default for ARM Compiler 6 is <code>-O0</code> .
<code>-Ospace</code>	<code>-Oz / -Os</code>	Performs optimizations to reduce image size at the expense of a possible increase in execution time.
<code>-Otime</code>	Default.	Performs optimizations to reduce execution time at the expense of a possible increase in image size. There is no equivalent of the <code>-Otime</code> option. ARM Compiler 6 optimizes for execution time by default, unless you specify the <code>-Os</code> or <code>-Oz</code> options.
<code>-S</code>	<code>-S</code>	Outputs the disassembly of the machine code generated by the compiler. The output from this option differs between releases. Older ARM Compiler versions produce output with <code>armasm</code> syntax while ARM Compiler 6 produces output with GNU syntax.
<code>--show_cmdline</code>	<code>-v</code>	Shows how the compiler processes the command line. The commands are shown normalized, and the contents of any via files are expanded.
<code>--vectorize</code>	<code>-fvectorize</code>	Enables the generation of Advanced SIMD vector instructions directly from C or C++ code.
<code>--via</code>	<code>@file</code>	Reads an additional list of compiler options from a file.
<code>--vsn</code>	<code>--version</code>	Displays version information and license details.

Related information

[The LLVM Compiler Infrastructure Project.](#)

2.2 Command-line options for preprocessing assembly source code

The version of `armasm` supplied with ARM Compiler 6 does not support the `--cpreproc` and `--cpreproc_opts` command-line options that were used in earlier versions of `armasm` to preprocess assembly source code.

If you are using `armasm` to assemble source code that requires the use of the preprocessor, you must first preprocess the code using `armclang`, then pipe it into `armasm`.

The following example shows the options required to preprocess and assemble the file `example.s`:

```
armclang --target=arm-arm-eabi-none -march=armv8-a -E -x assembler-with-cpp example.s |  
armasm --cpu=8-A.32 -o example.o -
```

Selected command-line options used in this example are:

- `--target=arm-arm-eabi-none -march=armv8-a`
Specifies that `armclang` targets ARMv8-A, AArch32 state.
- `--cpu=8-A.32`
Specifies that `armasm` targets ARMv8-A, AArch32 state.
- `-E`
Specifies that `armclang` only performs preprocessing on the file.
- `-x assembler-with-cpp`
Specifies that `armclang` handles the supplied source file as an assembly source file that requires preprocessing.
- `-`
Specifies that `armasm` reads the assembly source from `stdin` rather than from a file.

————— **Note** —————

Ensure that you specify compatible architectures in the `armclang --target` option and the `armasm --cpu` option.

Chapter 3

Compiler Source Code Compatibility

Provides details of source code compatibility between ARM Compiler 6 and older armcc compiler versions.

It contains the following sections:

- *3.1 Language extension compatibility: keywords* on page 3-17.
- *3.2 Language extension compatibility: attributes* on page 3-18.
- *3.3 Language extension compatibility: pragmas* on page 3-19.
- *3.4 Diagnostics for pragma compatibility* on page 3-22.
- *3.5 C and C++ implementation compatibility* on page 3-24.
- *3.6 Compatibility of C++ objects* on page 3-26.

3.1 Language extension compatibility: keywords

ARM Compiler 6 provides support for some keywords that were supported in older armcc compiler versions. Other keywords are not supported, or must be replaced with alternatives.

The following table lists some of the commonly used keywords that are supported by older versions of the compiler but are not supported by ARM Compiler 6. Replace any instances of these keywords in your code with the recommended alternative.

————— **Note** —————

This is not an exhaustive list of all unsupported keywords.

Table 3-1 Keyword language extensions that must be replaced

Keyword supported by older compiler versions	Recommended ARM Compiler 6 alternative
<code>__align(x)</code>	<code>__attribute__((aligned(x)))</code>
<code>__clz</code>	Use an inline CLZ assembly instruction or equivalent routine.
<code>__const</code>	<code>__attribute__((const))</code>
<code>__forceinline</code>	<code>__attribute__((always_inline))</code>
<code>__ldrex</code>	Use an inline LDREX assembly instruction.
<code>__packed</code>	<code>__attribute__((packed, aligned(1)))</code>
<code>__pure</code>	<code>__attribute__((pure))</code>
<code>__rev</code>	Use an inline REV assembly instruction.
<code>__sev</code>	Use an inline SEV assembly instruction.
<code>__softfp</code>	<code>__attribute__((__pcs__("aapcs")))</code>
<code>__strex</code>	Use an inline STREX assembly instruction.
<code>__weak</code>	<code>__attribute__((weak))</code>
<code>__wfe</code>	Use an inline WFE assembly instruction.

Related references

[3.5 C and C++ implementation compatibility on page 3-24.](#)

[3.2 Language extension compatibility: attributes on page 3-18.](#)

[3.3 Language extension compatibility: pragmas on page 3-19.](#)

3.2 Language extension compatibility: attributes

ARM Compiler 6 provides support for some function, variable, and type attributes that were supported in older armcc compiler versions. Other attributes are not supported.

The following attributes are supported by older compiler versions and ARM Compiler 6. These attributes do not require modification in your code:

- `__attribute__((aligned(x)))`
- `__attribute__((always_inline))`
- `__attribute__((const))`
- `__attribute__((deprecated))`
- `__attribute__((noinline))`
- `__declspec(noinline)`
- `__attribute__((nonnull))`
- `__attribute__((noreturn))`
- `__declspec(noreturn)`
- `__attribute__((nothrow))`
- `__declspec(nothrow)`
- `__attribute__((pcs("calling convention")))`
- `__attribute__((pure))`
- `__attribute__((section("name")))`

————— **Note** —————

Older compiler versions supported the `zero_init` attribute. ARM Compiler 6 does not support the `zero_init` attribute, but if the section name starts with `.bss.`, the variable is placed in a ZI section.

- `__attribute__((unused))`
- `__attribute__((used))`
- `__attribute__((visibility))`
- `__attribute__((weak))`
- `__attribute__((weakref))`

Related references

[3.5 C and C++ implementation compatibility](#) on page 3-24.

[3.1 Language extension compatibility: keywords](#) on page 3-17.

[3.3 Language extension compatibility: pragmas](#) on page 3-19.

3.3 Language extension compatibility: pragmas

ARM Compiler 6 provides support for some pragmas that were supported in older `armcc` compiler versions. Other pragmas are not supported, or must be replaced with alternatives.

The following table lists some of the commonly used pragmas that are supported by older versions of the compiler but are not supported by ARM Compiler 6. Replace any instances of these pragmas in your code with the recommended alternative.

Table 3-2 Pragma language extensions that must be replaced

Pragma supported by older <code>armcc</code> compiler versions	Recommended ARM Compiler 6 alternative
<code>#pragma import (symbol)</code>	<code>asm(" .global symbol\n")</code>
<code>#pragma anon_unions</code> <code>#pragma no_anon_unions</code>	<p>In C, anonymous structs and unions are a C11 extension which is enabled by default in <code>armclang</code>. If you specify the <code>-pedantic</code> option, the compiler emits warnings about extensions do not match the specified language standard. For example:</p> <pre>armclang --target=aarch64-arm-none-eabi -c -pedantic --std=c90 test.c test.c:3:5: warning: anonymous structs are a C11 extension [-Wc11-extensions]</pre> <p>In C++, anonymous unions are part of the language standard, and are always enabled. However, anonymous structs and classes are an extension. If you specify the <code>-pedantic</code> option, the compiler emits warnings about anonymous structs and classes. For example:</p> <pre>armclang --target=aarch64-arm-none-eabi -c -pedantic -xc++ test.c test.c:3:5: warning: anonymous structs are a GNU extension [-Wgnu-anonymous-struct]</pre> <p>Introducing anonymous unions, struct and classes using a <code>typedef</code> is a separate extension in <code>armclang</code>, which must be enabled using the <code>-fms-extensions</code> option.</p>
<code>#pragma arm</code> <code>#pragma thumb</code>	<code>armclang</code> does not support switching instruction set in the middle of a file. You can use the command line options <code>-marm</code> and <code>-mthumb</code> to specify the instruction set of the whole file.
<code>#pragma arm section</code>	<code>armclang</code> does not support setting the sections to be used for code, rodata, rwdata and zidata for the rest of the file. However the <code>__attribute__((section("name")))</code> attribute can be used to set the section of individual functions and variables.
<code>#pragma diag_default</code> <code>#pragma diag_suppress</code> <code>#pragma diag_remark</code> <code>#pragma diag_warning</code> <code>#pragma diag_error</code>	<p>The following pragmas provide equivalent functionality for <code>diag_suppress</code>, <code>diag_warning</code>, and <code>diag_error</code>:</p> <ul style="list-style-type: none"> <code>#pragma clang diagnostic ignored "-Wmultichar"</code> <code>#pragma clang diagnostic warning "-Wmultichar"</code> <code>#pragma clang diagnostic error "-Wmultichar"</code> <p>Note that these pragmas use <code>armclang</code> diagnostic groups, which do not have a precise mapping to <code>armcc</code> diagnostic tags.</p> <p><code>armclang</code> has no equivalent to <code>diag_default</code> or <code>diag_remark</code>. <code>diag_default</code> can be replaced by wrapping the change of diagnostic level with <code>#pragma clang diagnostic push</code> and <code>#pragma clang diagnostic pop</code>, or by manually returning the diagnostic to the default level.</p> <p>There is an additional diagnostic level supported in <code>armclang</code>, <code>fatal</code>, which causes compilation to fail without processing the rest of the file. You can set this as follows:</p> <pre>#pragma clang diagnostic fatal "-Wmultichar"</pre>

Table 3-2 Pragma language extensions that must be replaced (continued)

Pragma supported by older armcc compiler versions	Recommended ARM Compiler 6 alternative
#pragma exceptions_unwind	armclang does not support these pragmas.
#pragma no_exceptions_unwind	Use the <code>__attribute__((nothrow))</code> function attribute instead.
#pragma GCC system_header	This pragma is supported by both armcc and armclang, but <code>#pragma clang system_header</code> is the preferred spelling in armclang for new code.
#pragma hdrstop	armclang does not support these pragmas.
#pragma no_pch	
#pragma inline	armclang does not support these pragmas. However, inlining can be disabled on a per-function basis using the <code>__attribute__((noinline))</code> function attribute.
#pragma no_inline	The default behavior of both armcc and armclang is to inline functions when the compiler considers this worthwhile, and this is the behavior selected by using <code>#pragma inline</code> in armcc. To force a function to be inlined in armclang, use the <code>__attribute__((always_inline))</code> function attribute.
#pragma Onum	armclang does not support changing optimization options within a file. Instead these must be set on a per-file basis using command-line options.
#pragma Ospace	
#pragma Otime	
#pragma once	armclang supports this pragma.
#pragma pack	armclang supports this pragma.
#pragma pop	armclang does not support these pragmas.
#pragma push	If these are only used to control emission of diagnostics, <code>#pragma clang diagnostic push</code> and <code>#pragma clang diagnostic pop</code> can be used to achieve the same effect.
#pragma softfp_linkage	armclang does not support this pragma. Instead, use the <code>__attribute__((pcs("aapcs")))</code> function attribute to set the calling convention on a per-function basis, or use the <code>-mfloat-abi=soft</code> command-line option to set the calling convention on a per-file basis.
#pragma no_softfp_linkage	armclang does not support this pragma. Instead, use the <code>__attribute__((pcs("aapcs-vfp")))</code> function attribute to set the calling convention on a per-function basis, or use the <code>-mfloat-abi=hard</code> command-line option to set the calling convention on a per-file basis.
#pragma unroll[(n)]	armclang supports these pragmas.
#pragma unroll_completely	The default for <code>#pragma unroll</code> (that is, with no iteration count specified) differs between armclang and armcc: <ul style="list-style-type: none"> • With armclang, the default is to fully unroll a loop. • With armcc, the default is <code>#pragma unroll(4)</code>.
#pragma weak	armclang supports this pragma.

Related references

[3.5 C and C++ implementation compatibility on page 3-24.](#)

[3.1 Language extension compatibility: keywords on page 3-17.](#)

[3.2 Language extension compatibility: attributes on page 3-18.](#)

[3.4 Diagnostics for pragma compatibility on page 3-22.](#)

Related information

armclang Reference Guide: #pragma GCC system_header.

armclang Reference Guide: #pragma once.

armclang Reference Guide: #pragma pack(n).

armclang Reference Guide: #pragma weak symbol, #pragma weak symbol1 = symbol2.

armclang Reference Guide: #pragma unroll[(n)], #pragma unroll_completely.

3.4 Diagnostics for pragma compatibility

Older armcc compiler versions supported many pragmas which are not supported by armclang, but which could change the semantics of code. When armclang encounters these pragmas, it generates diagnostic messages.

The following table shows which diagnostics are generated for each pragma type, and the diagnostic group to which that diagnostic belongs. armclang generates diagnostics as follows:

- Errors indicate use of an armcc pragma which could change the semantics of code.
- Warnings indicate use of any other armcc pragma which is ignored by armclang.
- Pragmas other than those listed are silently ignored.

Table 3-3 Pragma diagnostics

Pragma supported by older compiler versions	Default diagnostic type	Diagnostic group
#pragma anon_unions	Warning	armcc-pragma-anon-unions
#pragma no_anon_unions	Warning	armcc-pragma-anon-unions
#pragma arm	Error	armcc-pragma-arm
#pragma arm section [<i>section_type_List</i>]	Error	armcc-pragma-arm
#pragma diag_default <i>tag[,tag,...]</i>	Error	armcc-pragma-diag
#pragma diag_error <i>tag[,tag,...]</i>	Error	armcc-pragma-diag
#pragma diag_remark <i>tag[,tag,...]</i>	Warning	armcc-pragma-diag
#pragma diag_suppress <i>tag[,tag,...]</i>	Warning	armcc-pragma-diag
#pragma diag_warning <i>tag[,tag,...]</i>	Warning	armcc-pragma-diag
#pragma exceptions_unwind	Error	armcc-pragma-exceptions-unwind
#pragma no_exceptions_unwind	Error	armcc-pragma-exceptions-unwind
#pragma GCC system_header	None	-
#pragma hdrstop	Warning	armcc-pragma-hdrstop
#pragma import <i>symbol_name</i>	Error	armcc-pragma-import
#pragma inline	Warning	armcc-pragma-inline
#pragma no_inline	Warning	armcc-pragma-inline
#pragma no_pch	Warning	armcc-pragma-no-pch
#pragma <i>Onum</i>	Warning	armcc-pragma-optimization
#pragma once	None	-
#pragma <i>Ospace</i>	Warning	armcc-pragma-optimization
#pragma <i>Otime</i>	Warning	armcc-pragma-optimization
#pragma pack	None	-
#pragma pop	Error	armcc-pragma-push-pop
#pragma push	Error	armcc-pragma-push-pop
#pragma softfp_linkage	Error	armcc-pragma-softfp-linkage
#pragma no_softfp_linkage	Error	armcc-pragma-softfp-linkage
#pragma thumb	Error	armcc-pragma-thumb

Table 3-3 Pragma diagnostics (continued)

Pragma supported by older compiler versions	Default diagnostic type	Diagnostic group
#pragma weak <i>symbol</i>	None	-
#pragma weak <i>symbol1</i> = <i>symbol2</i>	None	-

In addition to the above diagnostic groups, there are the following additional diagnostic groups:

armcc-pragmas

Contains all of the above diagnostic groups.

unknown-pragmas

Contains diagnostics about pragmas which are not known to armclang, and are not in the above table.

pragmas

Contains all pragma-related diagnostics, including armcc-pragmas and unknown-pragmas.

Any non-fatal armclang diagnostic group can be ignored, upgraded, or downgraded using the following command-line options:

Suppress a group of diagnostics:

-Wno-*diag-group*

Upgrade a group of diagnostics to warnings:

-W*diag-group*

Upgrade a group of diagnostics to errors:

-Werror=*diag-group*

Downgrade a group of diagnostics to warnings:

-Wno-error=*diag-group*

Related references

[3.3 Language extension compatibility: pragmas on page 3-19.](#)

3.5 C and C++ implementation compatibility

ARM Compiler 6 C and C++ implementation details differ from previous compiler versions.

The following table describes the C and C++ implementation detail differences.

Table 3-4 C and C++ implementation detail differences

Feature	Older versions of ARM Compiler	ARM Compiler 6
<i>Integer operations</i>		
Shifts	int shifts > 0 && < 127 int left shifts > 31 == 0 int right shifts > 31 == 0 (for unsigned or +ve), -1 (for -ve) long long shifts > 0 && < 63	Warns when shift amount > width of type. You can use the <code>-Wshift-count-overflow</code> option to suppress this warning.
Integer division	Checks that the sign of the remainder matches the sign of the numerator	The sign of the remainder is not necessarily the same as the sign of the numerator.
<i>Floating-point operations</i>		
Default standard	IEEE 754 standard, rounding to nearest representable value, exceptions disabled by default.	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. This is equivalent to the <code>--fpmode=ieee_full</code> option in older versions of ARM Compiler.
<i>Unions, enums and structs</i>		
Enum packing	Enums are implemented in the smallest integral type of the correct sign to hold the range of the enum values, unless <code>--enum_is_int</code> is specified in C++ mode.	By default enums are implemented as <code>int</code> , with <code>long long</code> used when required.
Signedness of plain bit-fields	Unsigned. Plain bit-fields declared without either the <code>signed</code> or <code>unsigned</code> qualifiers default to <code>unsigned</code> . The <code>--signed_bitfields</code> option treats plain bit-fields as <code>signed</code> .	Signed. Plain bit-fields declared without either the <code>signed</code> or <code>unsigned</code> qualifiers default to <code>signed</code> . There is no equivalent to either the <code>--signed_bitfields</code> or <code>--no_signed_bitfields</code> options.
<i>Misc C</i>		

Table 3-4 C and C++ implementation detail differences (continued)

Feature	Older versions of ARM Compiler	ARM Compiler 6
Tentative definitions (declarations of variables with no storage class and no initializer)	By default, <code>armcc</code> generates individual zero-initialized definitions for tentative definitions. These zero-initialized definitions are placed in a ZI section in the generated object. Multiple definitions in different files cause a <code>L6200E: Symbol multiply defined</code> linker error because the individual definitions clash with each other. The <code>armcc --relaxed_ref_def</code> option places the zero-initialized definitions in a common block. This common definition is not associated with any particular section or object, so multiple definitions resolve to a single definition at link time.	By default, <code>armclang</code> generates common zero-initialized definitions for tentative definitions. This is equivalent to the <code>armcc --relaxed_ref_def</code> option. The <code>armclang -fno-common</code> option produces individual zero-initialized definitions. This is equivalent to the default <code>armcc</code> behavior.
<code>sizeof(wchar_t)</code>	2 bytes	4 bytes
<i>Misc C++</i>		
Default C++ library	Rogue Wave Standard C++ Library	libc++ ————— Note ————— <ul style="list-style-type: none"> • The Rogue Wave C++ Library is incompatible with libc++. • There is limited compatibility between objects created with ARM Compiler 6 and those created with ARM Compiler 5.
Implicit inclusion	If compilation requires a template definition from a template declared in a header file <code>xyz.h</code> , the compiler implicitly includes the file <code>xyz.cc</code> or <code>xyz.CC</code> .	Not supported.
Alternative template lookup algorithms	When performing referencing context lookups, name lookup matches against names from the instantiation context as well as from the template definition context.	Not supported.
Exceptions	Off by default, function unwinding on with <code>--exceptions</code> by default.	On by default in C++ mode. Only supported with libc++.

Related references

- [3.1 Language extension compatibility: keywords on page 3-17.](#)
- [3.2 Language extension compatibility: attributes on page 3-18.](#)
- [3.3 Language extension compatibility: pragmas on page 3-19.](#)
- [3.6 Compatibility of C++ objects on page 3-26.](#)

3.6 Compatibility of C++ objects

The compatibility of C++ objects compiled with ARM Compiler 5 depends on the C++ libraries used.

Compatibility with objects compiled using Rogue Wave standard library headers

ARM Compiler 6 does not support binary compatibility with objects compiled using the Rogue Wave standard library include files.

There is no warning at link time when objects are mixed. The impact of mixing objects that have been compiled against different C++ standard library headers might include:

- Undefined symbol errors.
- Increased code size.
- Possible runtime errors.

Compatibility with C++ objects compiled using ARM Compiler 5

ARM Compiler 5 objects that use C++ but do not make use of the Rogue Wave header files can be compatible with ARM Compiler 6 objects that use `libc++`. However the presence of ARM Compiler 5 C++ objects causes the default library choice of `armLink` to change to Rogue Wave. Use the `-stdlib=libc++` option to force the selection of `libc++`.

For example, assuming `5.cpp` has no Rogue Wave library dependencies and `6.cpp` has `libc++` dependencies:

```
armcc -c 5.cpp
armclang -c 6.cpp
armLink -stdlib=libc++ 5.o 6.o -o foo.axf
```

By default, `armLink` would select Rogue Wave libraries because `5.o` is a C++ object compiled using ARM Compiler 5. The `-stdlib=libc++` option forces `armLink` to select `libc++` instead.

Related information

[armLink User Guide: `--stdlib`.](#)

[armclang Reference Guide: `--stdlib`.](#)

Chapter 4

Migrating ARM syntax assembly code to GNU syntax

Describes how to migrate assembly code from the legacy ARM syntax (used by `armasm`) to GNU syntax (used by `armclang`).

It contains the following sections:

- [4.1 Overview of differences between ARM and GNU syntax assembly code](#) on page 4-28.
- [4.2 Comments](#) on page 4-30.
- [4.3 Labels](#) on page 4-31.
- [4.4 Numeric local labels](#) on page 4-32.
- [4.5 Functions](#) on page 4-34.
- [4.6 Sections](#) on page 4-35.
- [4.7 Symbol naming rules](#) on page 4-36.
- [4.8 Numeric literals](#) on page 4-37.
- [4.9 Operators](#) on page 4-38.
- [4.10 Alignment](#) on page 4-39.
- [4.11 PC-relative addressing](#) on page 4-40.
- [4.12 Conditional directives](#) on page 4-41.
- [4.13 Data definition directives](#) on page 4-42.
- [4.14 Instruction set directives](#) on page 4-43.
- [4.15 Miscellaneous directives](#) on page 4-44.
- [4.16 Symbol definition directives](#) on page 4-45.

4.1 Overview of differences between ARM and GNU syntax assembly code

armasm (for assembling legacy assembly code) uses ARM syntax assembly code.

armclang aims to be compatible with GNU syntax assembly code (that is, the assembly code syntax supported by the GNU assembler, `as`).

If you have legacy assembly code that you want to assemble with `armclang`, you must convert that assembly code from ARM syntax to GNU syntax.

The specific instructions in your assembly code will not change during this migration process (although in some cases, the order of operands may change).

However, you need to make changes to the syntax of your assembly code. These changes include:

- The directives in your code.
- The format of labels, comments, and some types of literals.
- Some symbol names.
- The operators in your code.

The following examples show simple, equivalent, assembly code in both ARM and GNU syntax.

ARM syntax

```
; Simple ARM syntax example
;
; Iterate round a loop 10 times, adding 1 to a register each time.

        AREA ||.text||, CODE, READONLY, ALIGN=2

main PROC
    MOV    w5,#0x64      ; W5 = 100
    MOV    w4,#0        ; W4 = 0
    B      test_loop    ; branch to test_loop
loop
    ADD    w5,w5,#1     ; Add 1 to W5
    ADD    w4,w4,#1     ; Add 1 to W4
test_loop
    CMP    w4,#0xa      ; if W4 < 10, branch back to loop
    BLT   loop
    ENDP

        END
```

GNU syntax

```
// Simple GNU syntax example 4.2 Comments on page 4-30
//
// Iterate round a loop 10 times, adding 1 to a register each time.

        .section .text,"x"    // 4.6 Sections on page 4-35
        .balign 4

main:
    MOV    w5,#0x64          // 4.3 Labels on page 4-31
    MOV    w4,#0            // W5 = 100 4.8 Numeric Literals on page 4-37
    B      test_loop        // W4 = 0
loop:
    ADD    w5,w5,#1         // branch to test_loop
    ADD    w4,w4,#1         // Add 1 to W5
    ADD    w4,w4,#1         // Add 1 to W4
test_loop:
    CMP    w4,#0xa          // Add 1 to W5
    BLT   loop             // Add 1 to W4
    .end                    // if W4 < 10, branch back to loop
                          // 4.15 Miscellaneous directives on page 4-44
```

Related references

[4.2 Comments on page 4-30.](#)

[4.3 Labels on page 4-31.](#)

[4.4 Numeric local labels on page 4-32.](#)

[4.5 Functions on page 4-34.](#)

[4.6 Sections on page 4-35.](#)

- 4.7 *Symbol naming rules* on page 4-36.
- 4.8 *Numeric literals* on page 4-37.
- 4.9 *Operators* on page 4-38.
- 4.10 *Alignment* on page 4-39.
- 4.11 *PC-relative addressing* on page 4-40.
- 4.12 *Conditional directives* on page 4-41.
- 4.13 *Data definition directives* on page 4-42.
- 4.14 *Instruction set directives* on page 4-43.
- 4.15 *Miscellaneous directives* on page 4-44.
- 4.16 *Symbol definition directives* on page 4-45.

4.2 Comments

A comment identifies text that the assembler ignores.

ARM syntax

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.

For example:

```
; This whole line is a comment
; As is this line

myProc: PROC
    MOV     r1, #16    ; Load R0 with 16
```

GNU syntax

GNU syntax assembly code provides two different methods for marking comments:

- The `/*` and `*/` markers identify multiline comments:

```
/* This is a comment
   that spans multiple
   lines */
```

- The `//` marker identifies the remainder of a line as a comment:

```
MOV R0,#16    // Load R0 with 16
```

Related information

[GNU Binutils - Using as: Comments.](#)

[armasm User Guide: Syntax of source lines in assembly language.](#)

4.3 Labels

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.

ARM syntax

A label is written as a symbol beginning in the first column. A label can appear either in a line on its own, or in a line with an instruction or directive. Whitespace separates the label from any following instruction or directive:

```
MOV R0,#16
loop SUB R0,R0,#1 ; "loop" is a label
CMP R0,#0
BGT loop
```

GNU syntax

A label is written as a symbol that either begins in the first column, or has nothing but whitespace between the first column and the label. A label can appear either in a line on its own, or in a line with an instruction or directive. A colon ":" follows the label (whitespace is allowed between the label and the colon):

```
MOV R0,#16
loop:          // "loop" label on its own line
SUB R0,R0,#1
CMP R0,#0
BGT loop

MOV R0,#16
loop: SUB R0,R0,#1 // "loop" label in a line with an instruction
CMP R0,#0
BGT loop
```

Related references

[4.4 Numeric local labels on page 4-32.](#)

Related information

[GNU Binutils - Using as: Labels.](#)

4.4 Numeric local labels

Numeric local labels are a type of label that you refer to by a number rather than by name. Unlike other labels, the same numeric local label can be used multiple times and the same number can be used for more than one numeric local label.

ARM syntax

A numeric local label is a number in the range 0-99, optionally followed by a scope name corresponding to a ROUT directive.

Numeric local labels follow the same syntax as all other labels.

Refer to numeric local labels using the following syntax:

```
 %[F|B][A|T]n[rouname]
```

Where:

- F and B instruct the assembler to search forwards and backwards respectively. By default, the assembler searches backwards first, then forwards.
- A and T instruct the assembler to search all macro levels or only the current macro level respectively. By default, the assembler searches all macros from the current level to the top level, but does not search lower level macros.
- *n* is the number of the numeric local label in the range 0-99.
- *rouname* is an optional scope label corresponding to a ROUT directive. If *rouname* is specified in either a label or a reference to a label, the assembler checks it against the name of the nearest preceding ROUT directive. If it does not match, the assembler generates an error message and the assembly fails.

For example, the following code implements an incrementing loop:

```

1      MOV     r4,#1      ; r4=1
      ADD     r4,r4,#1   ; Local label
      CMP     r4,#0x5    ; Increment r4
      BLT    %b1        ; if r4 < 5...
                        ; ..branch backwards to local label "1"
```

Here is the same example using a ROUT directive to restrict the scope of the local label:

```

routA  ROUT
1routA MOV     r4,#1      ; Start of "routA" scope
      MOV     r4,#1      ; r4=1
      ADD     r4,r4,#1   ; Local label
      CMP     r4,#0x9    ; Increment r4
      BLT    %b1routA   ; if r4 < 9...
                        ; ..branch backwards to local label "1routA"
routB  ROUT           ; ..branch backwards to local label "1"
                        ; Start of "routB" scope (and therefore end of "routA" scope)
```

GNU syntax

A numeric local label is a number in the range 0-99.

Numeric local labels follow the same syntax as all other labels.

Refer to numeric local labels using the following syntax:

```
 n{f|b}
```

Where:

- *n* is the number of the numeric local label in the range 0-99.
- f and b instruct the assembler to search forwards and backwards respectively. There is no default. You must specify one of f or b.

For example, the following code implements an incrementing loop:

```

1:     MOV     r4,#1      // r4=1
      ADD     r4,r4,#1   // Local label
                        // Increment r4
```

```
CMP    r4,#0x5    // if r4 < 5...  
BLT    1b         // ...branch backwards to local label "1"
```

Note

GNU syntax assembly code does not provide mechanisms for restricting the scope of local labels.

Related references

[4.3 Labels on page 4-31.](#)

Related information

GNU Binutils - Using as: Labels.

GNU Binutils - Using as: Local labels.

armasm User Guide: Labels.

armasm User Guide: Numeric local labels.

armasm User Guide: Syntax of numeric local labels.

armasm User Guide: ROUT.

4.5 Functions

Assemblers can identify the start of a function when producing DWARF call frame information for ELF.

ARM syntax

The `FUNCTION` directive marks the start of a function. `PROC` is a synonym for `FUNCTION`.

The `ENDFUNC` directive marks the end of a function. `ENDP` is a synonym for `ENDFUNC`.

For example:

```
myproc PROC
; Procedure body
ENDP
```

GNU syntax

Use the `.type` directive to identify symbols as functions. For example:

```
.type myproc, @function
myproc:
; Procedure body
```

Note

GNU syntax assembly code provides the `.func` and `.endfunc` directives. However, these are not supported by `armclang`.

Note

Functions must be typed to link properly.

Related information

GNU Binutils - Using as: .type.

armasm User Guide: FUNCTION or PROC.

armasm User Guide: ENDFUNC or ENDP.

4.6 Sections

Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.

ARM syntax

The `AREA` directive instructs the assembler to assemble a new code or data section.

Section attributes within the `AREA` directive provide information about the section. Available section attributes include the following:

- `CODE` specifies that the section contains machine instructions.
- `READONLY` specifies that the section must not be written to.
- `ALIGN=n` specifies that the section is aligned on a 2^n -byte boundary

For example:

```
AREA mysection, CODE, READONLY, ALIGN=3
```

Note

The `ALIGN` attribute does not take the same values as the `ALIGN` directive. `ALIGN=n` (the `AREA` attribute) aligns on a 2^n byte boundary. `ALIGN n` (the `ALIGN` directive) aligns on an n -byte boundary.

GNU syntax

The `.section` directive instructs the assembler to assemble a new code or data section.

Flags provide information about the section. Available section flags include the following:

- `x` specifies that the section is executable.
- `w` specifies that the section is writable.

For example:

```
.section mysection,"x"
```

Not all ARM syntax `AREA` attributes map onto GNU syntax `.section` flags. For example, the ARM syntax `ALIGN` attribute corresponds to the GNU syntax `.balign` directive, rather than a `.section` flag:

```
.section mysection,"x"
.balign 8
```

Related information

GNU Binutils - Using as: .section.

GNU Binutils - Using as: .align.

armasm User Guide: AREA.

4.7 Symbol naming rules

ARM syntax assembly code and GNU syntax assembly code use similar, but different naming rules for symbols.

Symbol naming rules which are common to both ARM syntax and GNU syntax include:

- Symbol names must be unique within their scope.
- Symbol names are case-sensitive, and all characters in the symbol name are significant.
- Symbols must not use the same name as built-in variable names or predefined symbol names.

Symbol naming rules which differ between ARM syntax and GNU syntax include:

- ARM syntax symbols must start with a letter or the underscore character "_".
- GNU syntax symbols must start with a letter, the underscore character "_", or a period ".".
- ARM syntax symbols use double bars to delimit symbol names containing non-alphanumeric characters (except for the underscore):

```
IMPORT ||Image$$ARM_LIB_STACKHEAP$$ZI$$Limit||
```

GNU syntax symbols do not require double bars:

```
.global Image$$ARM_LIB_STACKHEAP$$ZI$$Limit
```

Related information

GNU Binutils - Using as: Symbol Names.

armasm User Guide: Symbol naming rules.

4.8 Numeric literals

ARM syntax assembly and GNU syntax assembly provide different methods for specifying some types of numeric literal.

Implicit shift operations

ARM syntax assembly allows immediate values with an implicit shift operation. For example, the `MOVK` instruction takes a 16-bit operand with an optional left shift. `armasm` accepts the instruction `MOVK x1, #0x40000`, converting the operand automatically to `MOVK x1, #0x4, LSL #16`.

GNU syntax assembly expects immediate values to be presented as encoded. The instruction `MOVK x1, #0x40000` results in the following message: `error: immediate must be an integer in range [0, 65535]`.

Hexadecimal literals

ARM syntax assembly provides two methods for specifying hexadecimal literals, the prefixes `"#&"` and `"#0x"`.

For example, the following are equivalent:

```
ADD    r1, #0xAF
ADD    r1, #&AF
```

GNU syntax assembly only supports the `"#0x"` prefix for specifying hexadecimal literals. Convert any `"#&"` prefixes to `"#0x"`.

`n_base-n-digits` format

ARM syntax assembly lets you specify numeric literals using the following format:

`n_base-n-digits`

For example:

- `2_1101` is the binary literal 1101 (13 in decimal).
- `8_27` is the octal literal 27 (23 in decimal).

GNU syntax assembly does not support the `n_base-n-digits` format. Convert all instances to a supported numeric literal form.

For example, you could convert:

```
ADD    r1, 2_1101
```

to:

```
ADD    r1, #13
```

or:

```
ADD    r1, #0xD
```

Related information

[GNU Binutils - Using as: Integers.](#)

[armasm User Guide: Syntax of numeric literals.](#)

4.9 Operators

ARM syntax assembly and GNU syntax assembly provide different methods for specifying some operators.

The following table shows how to translate ARM syntax operators to GNU syntax operators.

Table 4-1 Operator translation

ARM syntax operator	GNU syntax operator
:OR:	
:AND:	&
:NOT:	~
:SHL:	<<
:SHR:	>>
:LOR:	
:LAND:	&&

Related information

GNU Binutils - Using as: Infix Operators.

armasm User Guide: Unary operators.

armasm User Guide: Shift operators.

armasm User Guide: Addition, subtraction, and logical operators.

4.10 Alignment

Data and code must be aligned to appropriate boundaries.

For example, The T32 pseudo-instruction ADR can only load addresses that are word aligned, but a label within T32 code might not be word aligned. You must use an alignment directive to ensure four-byte alignment of an address within T32 code.

An alignment directive aligns the current location to a specified boundary by padding with zeros or NOP instructions.

ARM syntax

ARM syntax assembly provides the `ALIGN n` directive, where *n* specifies the alignment boundary in bytes. For example, the directive `ALIGN 128` aligns addresses to 128-byte boundaries.

ARM syntax assembly also provides the `PRESERVE8` directive. The `PRESERVE8` directive specifies that the current file preserves eight-byte alignment of the stack. GNU syntax assembly does not provide an equivalent to this directive.

GNU syntax

GNU syntax assembly provides the `.balign n` directive, which uses the same format as `ALIGN`.

Convert all instances of `ALIGN n` to `.balign n`.

Note

GNU syntax assembly also provides the `.align n` directive. However, the format of *n* varies from system to system. The `.balign` directive provides the same alignment functionality as `.align` with a consistent behavior across all architectures.

Related information

GNU Binutils - Using as: .align.

GNU Binutils - Using as: .balign.

armasm User Guide: REQUIRE8 and PRESERVE8.

armasm User Guide: ALIGN.

4.11 PC-relative addressing

ARM syntax assembly and GNU syntax assembly provide different methods for performing PC-relative addressing.

ARM syntax

ARM syntax assembly provides the symbol {pc} to let you use the address of the program counter.

For example:

```
ADRP x0, {pc}
```

GNU syntax

GNU syntax assembly does not support the {pc} symbol. Instead, it uses the special dot "." character, as follows:

```
ADRP x0, .
```

Related information

GNU Binutils - Using as: The Special Dot Symbol.

armasm User Guide: Register-relative and PC-relative expressions.

4.12 Conditional directives

Conditional directives let you specify conditions that control whether or not to assemble a sequence of assembly code.

The following table shows how to translate ARM syntax conditional directives to GNU syntax directives:

Table 4-2 Conditional directive translation

ARM syntax directive	GNU syntax directive
IF	.if
IF :DEF:	.ifdef
IF :LNOT::DEF:	.ifndef
ELSE	.else
ELSEIF	.elseif
ENDIF	.endif

Related information

GNU Binutils - Using as: .if.

GNU Binutils - Using as: .else.

GNU Binutils - Using as: .elseif.

GNU Binutils - Using as: .endif.

armasm User Guide: IF, ELSE, ENDIF, and ELIF.

4.13 Data definition directives

Data definition directives allocate memory, define data structures, and set initial contents of memory.

The following table shows how to translate ARM syntax data definition directives to GNU syntax directives:

————— **Note** —————

This list only contains examples of common data definition assembly directives. It is not exhaustive.

Table 4-3 Data definition directives translation

ARM syntax directive	GNU syntax directive	Description
DCB	.byte	Allocate one-byte blocks of memory, and specify the initial contents.
DCW	.hword	Allocate two-byte blocks of memory, and specify the initial contents.
DCD	.word	Allocate four-byte blocks of memory, and specify the initial contents.
DCQ	.quad	Allocate eight-byte blocks of memory, and specify the initial contents.
SPACE	.space	Allocate a zeroed block of memory.

Related information

GNU Binutils - Using as: .byte.

GNU Binutils - Using as: .word.

GNU Binutils - Using as: .hword.

GNU Binutils - Using as: .quad.

GNU Binutils - Using as: .space.

4.14 Instruction set directives

Instruction set directives instruct the assembler to interpret subsequent instructions as either A32 or T32 instructions.

The following table shows how to translate ARM syntax instruction set directives to GNU syntax directives:

Table 4-4 Instruction set directives translation

ARM syntax directive	GNU syntax directive	Description
ARM or CODE32	.arm or .code32	Interpret subsequent instructions as A32 instructions.
THUMB or CODE16	.thumb or .code16	Interpret subsequent instructions as T32 instructions.

Related information

GNU Binutils - Using as: ARM Machine Directives.

armasm User Guide: ARM or CODE32 directive.

armasm User Guide: CODE16 directive.

armasm User Guide: THUMB directive.

4.15 Miscellaneous directives

Miscellaneous directives perform a range of different functions.

The following table shows how to translate ARM syntax miscellaneous directives to GNU syntax directives:

Table 4-5 Miscellaneous directives translation

ARM syntax directive	GNU syntax directive	Description
foo EQU 0x1C	.equ foo, 0x1C	Assigns a value to a symbol. Note the rearrangement of operands.
EXPORT StartHere GLOBAL StartHere	.global StartHere .type StartHere, @function	Declares a symbol that can be used by the linker (that is, a symbol that is visible to the linker). armasm automatically determines the types of exported symbols. However, armclang requires that you explicitly specify the types of exported symbols using the .type directive. If the .type directive is not specified, the linker outputs warnings of the form: Warning: L6437W: Relocation #RELA:1 in test.o(.text) with respect to <i>symbol</i> ... Warning: L6318W: test.o(.text) contains branch to a non-code symbol <i>symbol</i> .
GET file INCLUDE file	.include file	Includes a file within the file being assembled.
IMPORT foo	.global foo	Provides the assembler with a name that is not defined in the current assembly.
INCBIN	.incbin	Partial support, armclang does not fully support .incbin.
INFO <i>n</i> , "string"	.print "string"	The INFO directive supports diagnostic generation on either pass of the assembly (specified by <i>n</i>). The .print directive does not let you specify a particular pass.
ENTRY	armlink -- entry= <i>location</i>	The ENTRY directive declares an entry point to a program. armclang does not provide an equivalent directive. Use armlink --entry= <i>location</i> to specify the entry point directly to the linker, rather than defining it in the assembly code.
END	.end	Marks the end of the assembly file.

Related information

[GNU Binutils - Using as: .type.](#)

[GNU Binutils - Using as: .print.](#)

[GNU Binutils - Using as: .equ.](#)

[GNU Binutils - Using as: .global.](#)

[GNU Binutils - Using as: .include.](#)

[GNU Binutils - Using as: .incbin.](#)

[armasm User Guide: ENTRY.](#)

[armasm User Guide: END.](#)

[armasm User Guide: INFO.](#)

[armasm User Guide: EXPORT or GLOBAL.](#)

[armlink User Guide: --entry.](#)

4.16 Symbol definition directives

Symbol definition directives declare and set arithmetic, logical, or string variables.

The following table shows how to translate ARM syntax symbol definition directives to GNU syntax directives:

————— **Note** —————

This list only contains examples of common symbol definition directives. It is not exhaustive.

Table 4-6 Symbol definition directives translation

ARM syntax directive	GNU syntax directive	Description
LCLA var	.set var, 0	Declare a local arithmetic variable, and initialize its value to 0.
LCLL var	.set var, FALSE	Declare a local logical variable, and initialize its value to FALSE.
LCLS var	.set var, ""	Declare a local string variable, and initialize its value to a null string.
GBLA var	.global var .set var, 0	Declare a global arithmetic variable, and initialize its value to 0.
GBLL var	.global var .set var, FALSE	Declare a global logical variable, and initialize its value to FALSE.
GBLS var	.global var .set var, ""	Declare a global string variable, and initialize its value to a null string.
var SETA expr	.set var, expr	Set the value of an arithmetic variable.
var SETL expr	.set var, expr	Set the value of a logical variable.
var SETS expr	.set var, expr	Set the value of a string variable.
foo RN 11	foo .reg r11	Define an alias foo for register R11.
foo QN q5.I32	foo .qn q5.i32	Define an I32-typed alias foo for the quad-precision register Q5.
foo DN d2.I32	foo .dn d2.i32	Define an I32-typed alias foo for the double-precision register D2.

Related information

GNU Binutils - Using as: ARM Machine Directives.

GNU Binutils - Using as: .global.

GNU Binutils - Using as: .set.