

ARM® Compiler

Version 6.5

Getting Started Guide



ARM® Compiler

Getting Started Guide

Copyright © 2014-2016 ARM. All rights reserved.

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

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 © 2014-2016, 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 Getting Started Guide

Preface

<i>About this book</i>	9
------------------------------	---

Chapter 1

Overview of the ARM® Compiler 6 Toolchain

1.1	<i>About ARMv8 terminology</i>	1-12
1.2	<i>Support level definitions</i>	1-13
1.3	<i>About ARM® Compiler</i>	1-16
1.4	<i>Host platform support for ARM® Compiler</i>	1-18
1.5	<i>About the toolchain documentation</i>	1-19
1.6	<i>ARM® Compiler toolchain licensing</i>	1-20
1.7	<i>Standards compliance in ARM® Compiler</i>	1-21
1.8	<i>Compliance with the ABI for the ARM Architecture (Base Standard)</i>	1-22
1.9	<i>GCC compatibility provided by ARM® Compiler 6</i>	1-24
1.10	<i>Toolchain environment variables</i>	1-25
1.11	<i>ARM architectures supported by the toolchain</i>	1-27
1.12	<i>ARM® Compiler and virtual address space</i>	1-28
1.13	<i>Compilation tools command-line option rules</i>	1-29
1.14	<i>ARM® Compiler package structure</i>	1-30
1.15	<i>Compiler command-line options</i>	1-31
1.16	<i>Clang and LLVM documentation</i>	1-33
1.17	<i>Further reading</i>	1-34

Chapter 2

Creating an Application

2.1	<i>Introduction to the ARM compilation tools</i>	2-37
-----	--	------

2.2	<i>The ARM compiler command</i>	2-38
2.3	<i>Building an image from C source</i>	2-39
2.4	<i>The ARM linker command</i>	2-40
2.5	<i>Linking an object file (armclang)</i>	2-41
2.6	<i>The ARM assembler commands</i>	2-42
2.7	<i>Building an image from GNU syntax assembly code</i>	2-43
2.8	<i>Building an image from legacy ARM syntax assembly code</i>	2-44
2.9	<i>The fromelf image converter command</i>	2-45

List of Figures

ARM® Compiler Getting Started Guide

<i>Figure 1-1</i>	<i>Integration boundaries in ARM Compiler 6.</i>	<i>1-14</i>
<i>Figure 2-1</i>	<i>A typical tool usage flow diagram</i>	<i>2-37</i>

List of Tables

ARM® Compiler Getting Started Guide

Table 1-1	Environment variables used by the toolchain	1-25
Table 1-2	Compiler command-line options	1-31

Preface

This preface introduces the *ARM® Compiler Getting Started Guide*.

It contains the following:

- [About this book](#) on page 9.

About this book

The ARM® Compiler Getting Started Guide provides general information for ARM Compiler 6 users.

Using this book

This book is organized into the following chapters:

Chapter 1 Overview of the ARM® Compiler 6 Toolchain

Gives general information about ARM® Compiler 6.

Chapter 2 Creating an Application

Describes how to create an application using ARM Compiler.

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 *ARM® Compiler Getting Started Guide*.
- The number ARM DUI0741F.
- 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 Information Center](#).
- [ARM Technical Support Knowledge Articles](#).
- [Support and Maintenance](#).
- [ARM Glossary](#).

Chapter 1

Overview of the ARM® Compiler 6 Toolchain

Gives general information about ARM® Compiler 6.

It contains the following sections:

- [1.1 About ARMv8 terminology](#) on page 1-12.
- [1.2 Support level definitions](#) on page 1-13.
- [1.3 About ARM® Compiler](#) on page 1-16.
- [1.4 Host platform support for ARM® Compiler](#) on page 1-18.
- [1.5 About the toolchain documentation](#) on page 1-19.
- [1.6 ARM® Compiler toolchain licensing](#) on page 1-20.
- [1.7 Standards compliance in ARM® Compiler](#) on page 1-21.
- [1.8 Compliance with the ABI for the ARM Architecture \(Base Standard\)](#) on page 1-22.
- [1.9 GCC compatibility provided by ARM® Compiler 6](#) on page 1-24.
- [1.10 Toolchain environment variables](#) on page 1-25.
- [1.11 ARM architectures supported by the toolchain](#) on page 1-27.
- [1.12 ARM® Compiler and virtual address space](#) on page 1-28.
- [1.13 Compilation tools command-line option rules](#) on page 1-29.
- [1.14 ARM® Compiler package structure](#) on page 1-30.
- [1.15 Compiler command-line options](#) on page 1-31.
- [1.16 Clang and LLVM documentation](#) on page 1-33.
- [1.17 Further reading](#) on page 1-34.

1.1 About ARMv8 terminology

ARMv8 introduces a number of terms to describe the state of the integer register bank and supported instruction sets.

ARMv8 introduces the following terms to describe the state of the integer register bank:

AArch32

The 32-bit Execution state in which the general-purpose register bank comprises 32-bit registers. There are 16 registers (R0 to R15) available in this state, including the PC. There are additional banked registers for different Processing Element (PE) modes.

AArch64

The 64-bit Execution state in which the general-purpose register bank comprises 64-bit registers. There are 31 registers (R0 to R30) available in this state, with register number 31 being a special case.

The general-purpose registers can be accessed as either 32-bit registers or 64-bit registers. The 32-bit registers are accessed using the W registers. The 64-bit registers are accessed using the X registers. The W registers access the lower half (bits 0 to 31) of the X registers.

Register number 31 represents:

Zero Register

In most cases register number 31 reads as zero when used as a source register, and discards the result when used as a destination register.

Stack Pointer

When used as a load/store base register, and in a small selection of arithmetic instructions, register number 31 provides access to the current stack pointer.

The PC is never accessible as a named register.

ARMv8 introduces the following terms to describe the instruction sets supported:

A32 instruction

An instruction that a processor can execute when in AArch32. When executing these instructions, the processor is in A32 state. These are 32-bit encoded instructions and must be word aligned. A32 instructions were previously called ARM instructions.

T32 instruction

An instruction that a processor can execute when in AArch32. When executing these instructions, the processor is in T32 state. These include 16-bit and 32-bit encoded instructions. These instructions must be half-word aligned. T32 instructions incorporate Thumb-2 technology. T32 instructions were previously called Thumb instructions.

A64 instruction

An instruction that a processor can execute when in AArch64. These are 32-bit encoded instructions and must be word aligned.

Note

The terms A32, T32, and AArch32 also apply to ARMv7 and ARMv6-M architectures, with the exception that the M-profile architectures do not support A32 instructions. In the context of instructions and states, the terms ARM and Thumb are the former terms for A32 and T32 respectively.

Note

Detailed information about the ARMv8 architecture is available under license. Contact your ARM Account Representative for details.

Related information

[ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.](#)

1.2 Support level definitions

Describes the levels of support for various ARM Compiler features.

ARM Compiler 6 is built on Clang and LLVM technology and as such, 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 <http://www.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 [BETA].

- 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 [ALPHA].

- 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 [COMMUNITY].

- 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 are going to 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 are going to 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:

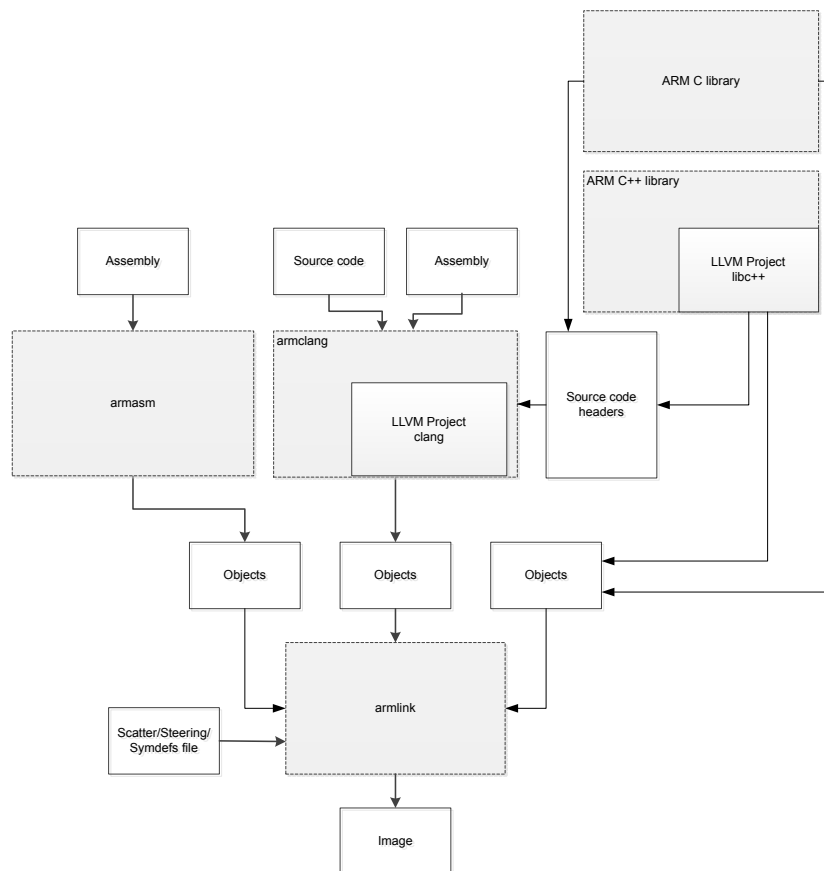


Figure 1-1 Integration boundaries in ARM Compiler 6.

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 1-13](#).

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.
- The ARM Compiler 6 `libc++` libraries do not support the Thread support library `<thread>`.
- 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](#).
- Use of C11 library features is unsupported.
- Any community feature that exclusively pertains to non-ARM architectures is not supported by ARM Compiler 6.
- Compilation for targets that implement architectures older than ARMv7 or ARMv6-M is not supported.

1.3 About ARM® Compiler

ARM Compiler enables you to build applications for the ARM family of processors from C, C++, or assembly language source.

The ARM Compiler toolchain comprises:

armclang

The compiler and assembler. This compiles C, C++, and GNU assembly language sources.

The compiler is based on LLVM and Clang technology.

LLVM is a set of open-source components that allow the implementation of optimizing compiler frameworks.

Clang is a compiler front end for LLVM, providing support for the C and C++ programming languages.

armasm

The legacy assembler. This assembles A32, A64, and T32 ARM syntax assembly code.

Only use **armasm** for legacy ARM syntax assembly code. Use the **armclang** assembler and GNU syntax for all new assembly files.

armlink

The linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program.

armar

The librarian. This enables sets of ELF object files to be collected together and maintained in archives or libraries. You can pass such a library or archive to the linker in place of several ELF files. You can also use the archive for distribution to a third party for further application development.

fromelf

The image conversion utility. This can also generate textual information about the input image, such as its disassembly and its code and data size.

————— **Note** —————

Disassembly is generated in ARM assembler syntax and not GNU assembler syntax.

libc++ and libc++abi libraries

The **libc++abi** library is a runtime library providing implementations of low-level language features.

The **libc++** library provides an implementation of the standard C++ library. It depends on the functions provided by **libc++abi**.

ARM C libraries

The ARM C libraries provide:

- An implementation of the library features as defined in the C standards.
- Nonstandard extensions common to many C libraries.
- POSIX extended functionality.
- Functions standardized by POSIX.

————— **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

ARM Compiler 6 is built on open-source technology. 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 open-source technology](#). For more information about the different support levels in ARM Compiler 6, see [Support level definitions on page 1-13](#).

Supporting software

You can debug the output from the toolchain with any debugger that is compatible with the ELF and DWARF specifications (DWARF 2, 3, or 4), such as ARM DS-5.

Check the ARM web site for updates and patches to the toolchain.

Related concepts

[2.2 The ARM compiler command on page 2-38.](#)

[2.4 The ARM linker command on page 2-40.](#)

[2.6 The ARM assembler commands on page 2-42.](#)

[2.9 The fromelf image converter command on page 2-45.](#)

1.4 Host platform support for ARM® Compiler

ARM Compiler 6 supports various Windows, Ubuntu, and Red Hat Enterprise Linux platforms. There is support for both 32-bit and 64-bit platforms.

See the [ARM® Compiler 6 Release Notes](#), for your version of the compiler, for a list of host platforms that ARM Compiler 6 supports.

Related concepts

[1.12 ARM® Compiler and virtual address space on page 1-28.](#)

1.5 About the toolchain documentation

ARM Compiler contains a suite of documents that describe how to use the tools and provide information on migration from, and compatibility with, earlier toolchain versions.

The toolchain documentation comprises:

Getting Started Guide (ARM DUI 0741) - this document

This document gives an overview of the toolchain.

Migration and Compatibility Guide (ARM DUI 0742)

This document describes the differences you must be aware of in ARM Compiler 6, when migrating your software from older versions of ARM compiler.

Software Development Guide (ARM DUI 0773)

This document contains information to help you develop code for various ARM architecture-based processors.

armclang Reference Guide (ARM DUI 0774)

This document provides user information for the ARM compiler, `armclang`. `armclang` is an optimizing C and C++ compiler that compiles Standard C and Standard C++ source code into machine code for ARM architecture-based processors.

armasm User Guide (ARM DUI 0801)

This document describes how to use the various features of the legacy ARM assembler, `armasm`. It also provides a detailed description of each assembler command-line option.

armlink User Guide (ARM DUI 0803)

This document describes how to use the various features of the linker, `armlink`. It also provides a detailed description of each linker command-line option.

armar User Guide (ARM DUI 0806)

This document describes how to use the various features of the librarian, `armar`. It also provides a detailed description of each `armar` command-line option.

fromelf User Guide (ARM DUI 0805)

This document describes how to use the various features of the ELF image converter, `fromelf`. It also provides a detailed description of each `fromelf` command-line option.

Errors and Warnings Reference Guide (ARM DUI 0807)

This document describes the errors and warnings that might be generated by each of the build tools in ARM Compiler toolchain.

ARM C and C++ Libraries and Floating-Point Support User Guide (ARM DUI 0808)

This document describes the features of the ARM C and C++ libraries, and how to use them. It also describes the floating-point support of the libraries.

Related concepts

[1.16 Clang and LLVM documentation on page 1-33.](#)

Related references

[1.17 Further reading on page 1-34.](#)

1.6 ARM® Compiler toolchain licensing

ARM Compiler requires a license.

Licensing of the ARM development tools is controlled by the FlexNet license management system.

ARMLMD_LICENSE_FILE specifies the location of your ARM license file. This environment variable must be set to point to your license server or license file.

Related references

[1.10 Toolchain environment variables](#) on page 1-25.

Related information

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

1.7 Standards compliance in ARM® Compiler

ARM Compiler conforms to the ISO C, ISO C++, ELF, and DWARF standards.

The level of compliance for each standard is:

ar

`armar` produces, and `armlink` consumes, UNIX-style object code archives. `armar` can list and extract most `ar`-format object code archives, and `armlink` can use an `ar`-format archive created by another archive utility providing it contains a symbol table member.

DWARF

The compiler generates DWARF 4 (DWARF Debugging Standard Version 4) debug tables with the `-g` option. The compiler can also generate DWARF 3 or DWARF 2 for backwards compatibility with legacy and third-party tools.

The linker and the `fromelf` utility can consume ELF format inputs containing DWARF 4, DWARF 3, and DWARF 2 format debug tables.

The legacy assembler `armasm` generates DWARF 3 debug tables with the `--debug` option. When assembling for AArch32, `armasm` can also generate DWARF 2 for backwards compatibility with legacy and third-party tools.

ISO C

The compiler accepts ISO C90, C99, and C11 source as input.

ISO C++

The compiler accepts ISO C++98 and C++11 source as input.

ELF

The toolchain produces relocatable and executable files in ELF format. The `fromelf` utility can translate ELF files into other formats.

Related concepts

[1.8 Compliance with the ABI for the ARM Architecture \(Base Standard\)](#) on page 1-22.

1.8 Compliance with the ABI for the ARM Architecture (Base Standard)

The ABI for the ARM Architecture (Base Standard) is a collection of standards. Some of these standards are open. Some are specific to the ARM architecture.

The *Application Binary Interface (ABI) for the ARM Architecture (Base Standard)* (BSABI) regulates the inter-operation of binary code and development tools in ARM architecture-based execution environments, ranging from bare metal to major operating systems such as ARM Linux.

By conforming to this standard, objects produced by the toolchain can work together with object libraries from different producers.

The BSABI consists of a family of specifications including:

AADWARF64

DWARF for the ARM 64-bit Architecture (AArch64). This ABI uses the DWARF 3 standard to govern the exchange of debugging data between object producers and debuggers. It also gives additional rules on how to use DWARF 3, and how it is extended in ways specific to the ARM 64-bit architecture.

AADWARF

DWARF for the ARM Architecture. This ABI uses the DWARF 3 standard to govern the exchange of debugging data between object producers and debuggers.

AAELF64

ELF for the ARM 64-bit Architecture (AArch64). This specification provides the processor-specific definitions required by ELF for AArch64-based systems. It builds on the generic ELF standard to govern the exchange of linkable and executable files between producers and consumers.

AAELF

ELF for the ARM Architecture. Builds on the generic ELF standard to govern the exchange of linkable and executable files between producers and consumers.

AAPCS64

Procedure Call Standard for the ARM 64-bit Architecture (AArch64). Governs the exchange of control and data between functions at runtime. There is a variant of the AAPCS for each of the major execution environment types supported by the toolchain.

AAPCS64 describes a number of different supported data models. ARM Compiler 6 implements the LP64 data model for AArch64 state.

AAPCS

Procedure Call Standard for the ARM Architecture. Governs the exchange of control and data between functions at runtime. There is a variant of the AAPCS for each of the major execution environment types supported by the toolchain.

BPABI

Base Platform ABI for the ARM Architecture. Governs the format and content of executable and shared object files generated by static linkers. Supports platform-specific executable files using post linking. Provides a base standard for deriving a platform ABI.

CLIBABI

C Library ABI for the ARM Architecture. Defines an ABI to the C library.

CPPABI64

C++ ABI for the ARM Architecture. This specification builds on the generic C++ ABI (originally developed for IA-64) to govern interworking between independent C++ compilers.

DBGOVL

Support for Debugging Overlaid Programs. Defines an extension to the ABI for the ARM Architecture to support debugging overlaid programs.

EHABI

Exception Handling ABI for the ARM Architecture. Defines both the language-independent and C++-specific aspects of how exceptions are thrown and handled.

RTABI

Run-time ABI for the ARM Architecture. Governs what independently produced objects can assume of their execution environments by way of floating-point and compiler helper function support.

If you are upgrading from a previous toolchain release, ensure that you are using the most recent versions of the ARM specifications.

Related concepts

[1.7 Standards compliance in ARM® Compiler](#) on page 1-21.

1.9 GCC compatibility provided by ARM® Compiler 6

The compiler in ARM Compiler 6 is based on Clang and LLVM technology. As such, it provides a high degree of compatibility with GCC.

ARM Compiler 6 can build the vast majority of C code that is written to be built with GCC. However, ARM Compiler is not 100% source compatible in all cases. Specifically, ARM Compiler does not aim to be bug-compatible with GCC. That is, ARM Compiler does not replicate GCC bugs.

1.10 Toolchain environment variables

Except for `ARMLMD_LICENSE_FILE`, ARM Compiler does not require any other environment variables to be set. However, there are situations where you might want to set environment variables.

The environment variables used by the toolchain are described in the following table.

Where an environment variable is identified as GCC compatible, the GCC documentation provides full information about that environment variable. See [Environment Variables Affecting GCC](#) on the [GCC web site](#).

Table 1-1 Environment variables used by the toolchain

Environment variable	Setting
<code>ARM_PRODUCT_PATH</code>	<p>Required only if you have a DS-5 toolkit license and you are running the ARM Compiler tools outside of the DS-5 environment.</p> <p>Use this environment variable to specify the location of the <code>sw/mappings</code> directory within an ARM Compiler or DS-5 installation.</p>
<code>ARM_TOOL_VARIANT</code>	<p>Required only if you have a DS-5 toolkit license and you are running the ARM Compiler tools outside of the DS-5 environment.</p> <p>If you have an ultimate license, set this environment variable to <code>ult</code> to enable the Ultimate features. See FAQ 16372 for more information.</p>
<code>ARMCOMPILER6_ASMOPT</code>	<p>An optional environment variable to define additional assembler options that are to be used outside your regular makefile.</p> <p>The options listed appear before any options specified for the <code>armasm</code> command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.</p>
<code>ARMCOMPILER6_CLANGOPT</code>	<p>An optional environment variable to define additional <code>armclang</code> options that are to be used outside your regular makefile.</p> <p>The options listed appear before any options specified for the <code>armclang</code> command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.</p>
<code>ARMCOMPILER6_FROMELFOPT</code>	<p>An optional environment variable to define additional <code>fromelf</code> image converter options that are to be used outside your regular makefile.</p> <p>The options listed appear before any options specified for the <code>fromelf</code> command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.</p>
<code>ARMCOMPILER6_LINKOPT</code>	<p>An optional environment variable to define additional linker options that are to be used outside your regular makefile.</p> <p>The options listed appear before any options specified for the <code>armlink</code> command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.</p>
<code>ARMROOT</code>	Your installation directory root, <i>install_directory</i> .

Table 1-1 Environment variables used by the toolchain (continued)

Environment variable	Setting
ARMLMD_LICENSE_FILE	<p>This environment variable must be set, and specifies the location of your ARM license file. See the ARM® DS-5 License Management Guide for information on this environment variable.</p> <hr/> <p>Note</p> <p>On Windows, the length of ARMLMD_LICENSE_FILE must not exceed 260 characters.</p> <hr/>
C_INCLUDE_PATH	<p>GCC compatible environment variable. Adds the specified directories to the list of places that are searched to find included C files.</p>
COMPILER_PATH	<p>GCC compatible environment variable. Adds the specified directories to the list of places that are searched to find subprograms.</p>
CPATH	<p>GCC compatible environment variable. Adds the specified directories to the list of places that are searched to find included files regardless of the source language.</p>
CPLUS_INCLUDE_PATH	<p>GCC compatible environment variable. Adds the specified directories to the list of places that are searched to find included C++ files.</p>
TMP	<p>Used on Windows platforms to specify the directory to be used for temporary files.</p>
TMPDIR	<p>Used on Red Hat Linux platforms to specify the directory to be used for temporary files.</p>

Related concepts

[1.6 ARM® Compiler toolchain licensing on page 1-20.](#)

Related information

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

1.11 ARM architectures supported by the toolchain

ARM Compiler supports a number of different architecture profiles.

ARM Compiler supports the following architectures:

- ARMv8-A bare metal targets.
- ARMv8.1-A bare metal targets.
- ARMv8.2-A bare metal targets.
- ARMv8-R targets.
- ARMv8-M targets.
- ARMv7-A bare metal targets.
- ARMv7-R targets.
- ARMv7-M targets.
- ARMv6-M targets.

When compiling code, the compiler needs to know which architecture to target in order to take advantage of features specific to that architecture.

To specify a target, you must supply the target execution state (AArch32 or AArch64), together with either a target architecture (for example ARMv8-A) or a target processor (for example Cortex-A53).

To specify a target execution state (AArch64 or AArch32) with `armclang`, use the mandatory `--target` command-line option:

```
--target=arch-vendor-os-abi
```

Supported targets include:

`aarch64-arm-none-eabi`

Generates A64 instructions for AArch64 state. Implies `-march=armv8-a` unless `-mcpu` is specified.

`arm-arm-none-eabi`

Generates A32/T32 instructions for AArch32 state. Must be used in conjunction with `-march` (to target an architecture) or `-mcpu` (to target a processor).

To generate generic code that runs on any processor with a particular architecture, use the `-march` option. Use the `-march=list` option to see all supported architectures.

To optimize your code for a particular processor, use the `-mcpu` option. Use the `-mcpu=list` option to see all supported processors.

Note

The `--target`, `-march`, `-mcpu`, and `-mfpu` options are `armclang` options. For all of the other tools, such as `armasm` and `armlink`, use the `--cpu` and `--fpu` options to specify target processors and architectures.

1.12 ARM® Compiler and virtual address space

ARM Compiler provides 64-bit versions of `armclang` and `armlink`.

`armasm`, `armar` and `fromelf` are only available as 32-bit applications. This limits the virtual address space and file size available to those tools. If these limits are exceeded, the tool reports an error message to indicate that there is not enough memory. This might cause confusion because sufficient physical memory is available but the application cannot access it.

The 64-bit `armclang` and `armlink` tools can utilize the greater amount of memory available on 64-bit host machines.

Related references

[1.4 Host platform support for ARM® Compiler on page 1-18.](#)

1.13 Compilation tools command-line option rules

You can control many aspects of the compilation tools operation with command-line options.

armclang option rules

armclang follows the same syntax rules as GCC. Some options are preceded by a single dash -, others by a double dash --. Some options require an = character between the option and the argument, others require a space character.

armasm, armar, armlink, and fromelf command-line syntax rules

The following rules apply, depending on the type of option:

Single-letter options

All single-letter options, including single-letter options with arguments, are preceded by a single dash -. You can use a space between the option and the argument, or the argument can immediately follow the option. For example:

```
armar -r -a obj1.o mylib.a obj2.o  
armar -r -aobj1.o mylib.a obj2.o
```

Keyword options

All keyword options, including keyword options with arguments, are preceded by a double dash --. An = or space character is required between the option and the argument. For example:

```
armlink myfile.o --cpu=list  
armlink myfile.o --cpu list
```

Command-line syntax rules common to all tools

To compile files with names starting with a dash, use the POSIX option -- to specify that all subsequent arguments are treated as filenames, not as command switches. For example, to link a file named -ifile_1, use:

```
armlink -- -ifile_1
```

In some Unix shells, you might have to include quotes when using arguments to some command-line options, for example:

```
armlink obj1.o --keep='s.o(vect)'
```

1.14 ARM® Compiler package structure

The structure of the ARM Compiler distribution package is as follows.

```
ARMCompiler6.4/  
|- bin/  
|- include/  
|   |- libcxx  
|- lib/  
|   |- armlib  
|   |- libcxx  
|- sw/  
|   |- info/  
|   |- mappings/
```

bin/

Contains all binary tools.

include/

Contains C and all other non-C++ system include files.

include/libcxx

Contains include files for the libc++ and libc++abi libraries.

lib/armlib

Contains ARM C, floating-point, math, and helper libraries.

lib/libcxx

Contains the libc++ and libc++abi library files.

sw/info/

Contains the README and information about third-party software licenses.

sw/mappings/

Contains license mapping files.

1.15 Compiler command-line options

Describes the most common ARM Compiler command-line options.

ARM Compiler provides many command-line options, including most Clang command-line options and several ARM-specific options. 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, see the Clang and LLVM documentation.

Table 1-2 Compiler command-line options

Option	Description
-c	Performs the compilation step, but not the link step.
-D	Defines a preprocessing macro.
-E	Executes only the preprocessor step.
-finline-functions	Enables inlining of functions.
-fvectorize	Enables the generation of Advanced SIMD vector instructions directly from C or C++ code.
-g	Generates DWARF debug tables.
-I	Adds the specified directories to the list of places that are searched to find included files.
-M	Instructs the compiler to produce a list of makefile dependency lines suitable for use by a make utility.
-march=name	Generates code for the specified architecture, for example <code>-march=armv8-a</code> or <code>-march=armv7-a</code> .
-march=list	Displays a list of all the supported architectures for your target.
-marm	Targets the A32 (ARMv8-A AArch32 state) or ARM (ARMv7 and earlier) instruction set. For example <code>--target=arm-arm-none-eabi -march=armv8-a -marm</code> . This is the default for all targets that support ARM or A32 instructions. The <code>-marm</code> option is not valid for AArch64 targets. The compiler ignores the <code>-marm</code> option and generates a warning for AArch64 targets.
-mcpu=name	Generates code for the specified processor, for example <code>-mcpu=cortex-a53</code> , <code>-mcpu=cortex-m4</code> , or <code>-mcpu=cortex-a15</code> .
-mcpu=list	Displays a list of all the supported processors for your target.
-mthumb	Targets the T32 (ARMv8-A AArch32 state) or Thumb (ARMv7 and earlier) instruction set. For example <code>--target=arm-arm-none-eabi -march=armv8-a -mthumb</code> . The <code>-mthumb</code> option is the default for all AArch32 targets that do not support ARM instructions, and is not valid for AArch64 targets. The compiler ignores the <code>-mthumb</code> option and generates a warning for AArch64 targets.
-o	Specifies the name of the output file.
-Onum	Specifies the level of performance optimization to be used when compiling source files, for example <code>-O0</code> or <code>-O3</code> .

Table 1-2 Compiler command-line options (continued)

Option	Description
-Oz / -Os	<p>Performs optimizations to reduce image size at the expense of a possible decrease in performance.</p> <p>-Os balances code size against performance. -Oz optimizes for code size.</p> <p>By default, the compiler performs optimizations to increase performance, at the expense of a possible increase in image size.</p>
-S	Outputs the disassembly of the machine code generated by the compiler.
-std= <i>name</i>	Compiles for the specified language standard, for example -std=c90 or -std=c++98.
--target= <i>arch-vendor-os-abi</i>	<p>Generates code for the selected execution state (AArch32 or AArch64), either --target=aarch64-arm-none-eabi or --target=arm-arm-none-eabi.</p>
-v	Shows how the compiler processes the command line. The commands are shown normalized, and the contents of any via files are expanded.
--vsn	Displays version information and license details.
-xc / -xc++	<p>Specifies the language of subsequent source files, interpreting all subsequent files as .c (-xc) or .cpp (-xc++) regardless of file extension.</p> <p>These are positional arguments and only affect subsequent input files on the command line.</p>
-Xlinker	Specifies a linker command-line option to pass to the linker when a link step is being performed after compilation.

1.16 Clang and LLVM documentation

ARM Compiler is based on Clang and LLVM compiler technology.

The ARM Compiler documentation describes features that are specific to, and supported by, ARM Compiler. Any features specific to ARM Compiler that are not documented are not supported and are used at your own risk. Although open-source Clang features are available, they are not supported by ARM and are used at your own risk. You are responsible for making sure that any generated code using unsupported features is operating correctly.

The *Clang Compiler User's Manual*, available from the LLVM Compiler Infrastructure Project web site <http://clang.llvm.org>, provides open-source documentation for Clang.

Related references

1.5 About the toolchain documentation on page 1-19.

1.17 Further reading on page 1-34.

Related information

The LLVM Compiler Infrastructure Project.

1.17 Further reading

Additional information on developing code for the ARM family of processors is available from both ARM and third parties.

ARM publications

ARM periodically provides updates and corrections to its documentation. See [ARM Infocenter](#) for current errata sheets and addenda, and the ARM Frequently Asked Questions (FAQs).

For full information about the base standard, software interfaces, and standards supported by ARM, see [Application Binary Interface \(ABI\) for the ARM Architecture](#).

In addition, see the following documentation for specific information relating to ARM products:

- [ARM Architecture Reference Manuals](#).
- [Cortex-A series processors](#).
- [Cortex-R series processors](#).
- [Cortex-M series processors](#).

Other publications

This ARM Compiler tools documentation is not intended to be an introduction to the C or C++ programming languages. It does not try to teach programming in C or C++, and it is not a reference manual for the C or C++ standards. Other publications provide general information about programming.

The following publications describe the C++ language:

- *ISO/IEC 14882:2014, C++ Standard*.
- Stroustrup, B., *The C++ Programming Language* (4th edition, 2013). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 978-0321563842.

The following publications provide general C++ programming information:

- Stroustrup, B., *The Design and Evolution of C++* (1994). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-54330-3.

This book explains how C++ evolved from its first design to the language in use today.

- Vandevoorde, D and Josuttis, N.M. *C++ Templates: The Complete Guide* (2003). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-73484-2.
- Meyers, S., *Effective C++* (3rd edition, 2005). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 978-0321334879.

This provides short, specific guidelines for effective C++ development.

- Meyers, S., *More Effective C++* (2nd edition, 1997). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-92488-9.

The following publications provide general C programming information:

- ISO/IEC 9899:2011, *C Standard*.

The standard is available from national standards bodies (for example, AFNOR in France, ANSI in the USA).

- Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* (2nd edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8.

This book is co-authored by the original designer and implementer of the C language, and is updated to cover the essentials of ANSI C.

- Harbison, S.P. and Steele, G.L., *A C Reference Manual* (5th edition, 2002). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-089592-X.

This is a very thorough reference guide to C, including useful information on ANSI C.

- Plauger, P., *The Standard C Library* (1991). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-131509-9.

This is a comprehensive treatment of ANSI and ISO standards for the C Library.

- Koenig, A., *C Traps and Pitfalls*, Addison-Wesley (1989), Reading, Mass. ISBN 0-201-17928-8.

This explains how to avoid the most common traps in C programming. It provides informative reading at all levels of competence in C.

See [The DWARF Debugging Standard web site](#) for the latest information about the *Debug With Arbitrary Record Format* (DWARF) debug table standards and ELF specifications.

Related concepts

[1.16 Clang and LLVM documentation on page 1-33.](#)

Related references

[1.5 About the toolchain documentation on page 1-19.](#)

Chapter 2

Creating an Application

Describes how to create an application using ARM Compiler.

It contains the following sections:

- [2.1 Introduction to the ARM compilation tools on page 2-37.](#)
- [2.2 The ARM compiler command on page 2-38.](#)
- [2.3 Building an image from C source on page 2-39.](#)
- [2.4 The ARM linker command on page 2-40.](#)
- [2.5 Linking an object file \(armclang\) on page 2-41.](#)
- [2.6 The ARM assembler commands on page 2-42.](#)
- [2.7 Building an image from GNU syntax assembly code on page 2-43.](#)
- [2.8 Building an image from legacy ARM syntax assembly code on page 2-44.](#)
- [2.9 The fromelf image converter command on page 2-45.](#)

2.1 Introduction to the ARM compilation tools

The compilation tools allow you to build executable images, partially linked object files, and shared object files, and to convert images to different formats.

A typical application development might involve the following:

- Developing C/C++ source code for the main application (`armclang`).
- Developing assembly source code for near-hardware components, such as interrupt service routines (`armclang`, or `armasm` for legacy assembly code).
- Linking all objects together to generate an image (`armlink`).
- Converting an image to flash format in plain binary, Intel Hex, and Motorola-S formats (`fromelf`).

The following figure shows how the compilation tools are used for the development of a typical application.

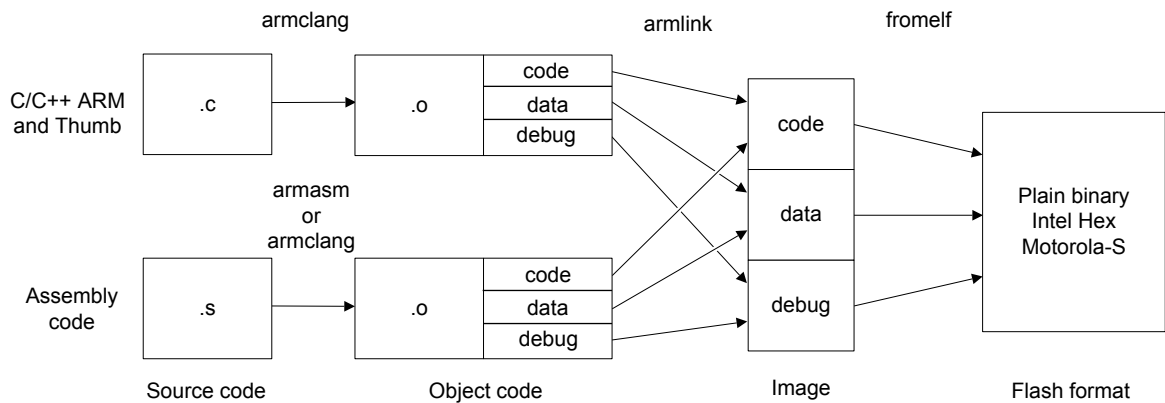


Figure 2-1 A typical tool usage flow diagram

Note

`armasm` is only supported for legacy assembly code. Use `armclang` for any new assembly code.

Related concepts

[2.2 The ARM compiler command on page 2-38.](#)

[2.4 The ARM linker command on page 2-40.](#)

[2.6 The ARM assembler commands on page 2-42.](#)

[2.9 The fromelf image converter command on page 2-45.](#)

2.2 The ARM compiler command

The compiler, `armclang`, can compile C and C++ source code into A32 instructions and T32 instructions for AArch32 targets, or A64 instructions for AArch64 state targets.

Typically, you invoke the compiler as follows:

```
armclang --target=target [options] file_1 ... file_n
```

You can specify one or more input files. The compiler produces one object file for each source input file.

2.3 Building an image from C source

This example shows how to build an image from C code with `armclang` and `armlink`.

To compile a C source file `hello_world.c`:

Procedure

1. Compile the C source with the following command:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c -O1 -o hello_world.o -xc -std=c90 -g hello_world.c
```

The following options are commonly used:

--target=arm-arm-none-eabi

Targets the AArch32 architecture profile.

-march=armv8-a

Targets the ARMv8-A architecture profile.

————— Note —————

For AArch32 targets (`--target=arm-arm-none-eabi`) there is no default architecture profile, so you must specify either `-march` (to target an architecture) or `-mcpu` (to target a processor). However for AArch64 targets (`--target=aarch64-arm-none-eabi`), ARMv8-A is the default architecture profile, and the `-march` option is not required.

-c

Tells the compiler to compile only, and not link.

-xc

Tells the compiler that the source is C code.

————— Note —————

The compiler infers the source code language from the filename extension. The `-x` option overrides the inferred language setting. The `-x` option is not required for this example, it is provided for illustrative purposes only.

-std=c90

Tells the compiler that the source is ISO C90 C code.

-O1

Tells the compiler to generate code with restricted optimizations applied to give a satisfactory debug view with good code density and performance.

-g

Tells the compiler to add debug tables for source-level debugging.

-o filename

Tells the compiler to create an object file with the specified *filename*.

2. Link the file:

```
armlink hello_world.o --info totals -o hello_world.axf
```

3. Use an ELF and DWARF 4 compatible debugger to load and run the image.

Related tasks

[2.5 Linking an object file \(armclang\) on page 2-41.](#)

2.4 The ARM linker command

The linker combines the contents of one or more object files with selected parts of one or more object libraries to produce an image or object file.

Typically, you invoke the linker as follows:

```
armlink [options] file_1 ... file_n
```

Related tasks

[2.5 Linking an object file \(armclang\) on page 2-41.](#)

Related information

[armlink Reference Guide.](#)

2.5 Linking an object file (armclang)

This example shows how to link an object file with `armlink`.

To link the object file `hello_world.o`, enter:

```
armlink --info totals -o hello_world.axf hello_world.o
```

where:

- `-o` Specifies the output file as `hello_world.axf`.
- `--info totals` Tells the linker to display totals of the Code and Data sizes for input objects and libraries.

Related concepts

[2.4 The ARM linker command on page 2-40.](#)

2.6 The ARM assembler commands

ARM Compiler provides two assemblers: `armclang` for GNU syntax assembly code, and `armasm` for legacy ARM syntax assembly code.

- `armclang` includes an assembler for GNU syntax assembly language source code. Use GNU syntax for all new assembly source code, and use `armclang` to assemble these source files.

Typically, you invoke the `armclang` assembler as follows:

```
armclang --target=target [options] file_1.s ... file_n.s
```

You can specify one or more `.s` input files. `armclang` automatically identifies from the `.s` suffix that the input file contains assembly code, and assembles the source files. `armclang` produces one object file for each source input file.

————— **Note** —————

The *GNU Binutils - Using `as`* documentation provides complete information about GNU syntax assembly code.

The *Migration and Compatibility Guide* contains detailed information about the differences between ARM and GNU syntax assembly to help you migrate legacy assembly code.

- `armasm` assembles ARM assembly code. You should only use `armasm` for legacy assembly files.

Typically, you invoke the `armasm` assembler as follows:

```
armasm --cpu=name [options] file_1.s ... file_n.s
```

Related tasks

[2.8 Building an image from legacy ARM syntax assembly code on page 2-44.](#)

Related information

[GNU Binutils - Using `as`.](#)

[Migrating ARM syntax assembly code to GNU syntax.](#)

2.7 Building an image from GNU syntax assembly code

This example shows how to build GNU syntax assembly language code with `armclang`.

Note

Use GNU syntax for all new assembly source code, and use `armclang` to assemble these source files. Only use ARM syntax and `armasm` for legacy assembly files.

To build an assembly program, for example `hello_world.s`:

Procedure

1. Assemble the source file:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 hello_world.s
```

`armclang` assembles the source file and automatically calls the linker to produce an executable image, `a.out`.

2. Use an ELF and DWARF 4 compatible debugger to load and run the image.
Step through the program and examine the registers to see how they change.

2.8 Building an image from legacy ARM syntax assembly code

This example shows how to build ARM assembly language code with `armasm`.

Note

Only use `armasm` for legacy assembly files. Use `armclang` to assemble new assembly files.

To build an assembly program, for example `hello_world.s`:

Procedure

1. Assemble the source file:

```
armasm --cpu=7-M --debug hello_world.s
```

2. Link the object file:

```
armlink hello_world.o -o hello_world.axf
```

Related concepts

[2.6 The ARM assembler commands on page 2-42.](#)

2.9 The fromelf image converter command

fromelf allows you to convert ELF files into different formats and display information about them.

The features of the fromelf image converter include:

- Converting an executable image in ELF executable format to other file formats.
- Controlling debug information in output files.
- Disassembling either an ELF image or an ELF object file. Disassembly is generated in ARM assembler syntax and not GNU assembler syntax.
- Protecting *intellectual property* (IP) in images and objects that are delivered to third parties.
- Printing information about an ELF image or an ELF object file.

Example 2-1 Examples

The following examples show how to use fromelf:

```
fromelf --text -c -s --output=outfile.lst infile.axf
```

Creates a plain text output file that contains the disassembled code and the symbol table of an ELF image.

```
fromelf --bin --16x2 --output=outfile.bin infile.axf
```

Creates two files in binary format (outfile0.bin and outfile1.bin) for a target system with a memory configuration of a 16-bit memory width in two banks.

The output files in the last example are suitable for writing directly to a 16-bit Flash device.
