Arm® Compiler

Version 6.9

User Guide

arm

100748_0609_00_en

Arm® Compiler

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0606-00	04 November 2016	Non-Confidential	Arm Compiler v6.6 Release
0607-00	05 April 2017	Non-Confidential	Arm Compiler v6.7 Release
0608-00	30 July 2017	Non-Confidential	Arm Compiler v6.8 Release.
0609-00	25 October 2017	Non-Confidential	Arm Compiler v6.9 Release.

Non-Confidential Proprietary Notice

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 User Guide

List of Figures
Arm® Compiler User Guide

List of Tables
Arm® Compiler User Guide

Preface

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

It contains the following:

- *About this book* on page 10.

About this book

The Arm® Compiler User Guide provides information for users new to Arm Compiler 6.

Using this book

This book is organized into the following chapters:

Chapter 1 Getting Started

This introduces Arm Compiler 6 and helps you start working with Arm Compiler 6 quickly. You can use Arm Compiler 6 from Arm DS-5, Keil MDK, or as a standalone product.

Chapter 2 Using Common Compiler Options

There are many options that you can use to control how Arm Compiler 6 generates code for your application. This section lists the mandatory and commonly used optional command-line arguments, such as to control target selection, optimization, and debug view.

Chapter 3 Writing Optimized Code

To make best use of the optimization capabilities of Arm Compiler, there are various options, pragmas, attributes, and coding techniques that you can use.

Chapter 4 Using Assembly and Intrinsics in C or C++ Code

All code for a single application can be written in the same source language. This is usually a high-level language such as C or C++ that is compiled to instructions for Arm architectures. However, in some situations you might need lower-level control than what C and C++ provide.

Chapter 5 Mapping Code and Data to the Target

There are various options in Arm Compiler to control how code, data and other sections of the image are mapped to specific locations on the target.

Chapter 6 Embedded Software Development

Describes how to develop embedded applications with Arm Compiler, with or without a target system present.

Appendix A Supporting reference information

The various features in Arm Compiler might have different levels of support, ranging from fully supported product features to community features.

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.

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

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

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

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

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

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

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

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

- The title *Arm Compiler User Guide*.
- The number 100748_0609_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Other information

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

Chapter 1
Getting Started

This introduces Arm Compiler 6 and helps you start working with Arm Compiler 6 quickly. You can use Arm Compiler 6 from Arm DS-5, Keil MDK, or as a standalone product.

It contains the following sections:

1.1 Introduction to Arm® Compiler 6

Arm Compiler 6 is the most advanced C and C++ compilation toolchain from Arm for Arm Cortex® processors. Arm Compiler 6 is developed alongside the Arm architecture, and is therefore tuned to generate highly efficient code for embedded bare-metal applications ranging from small sensors to 64-bit devices.

Arm Compiler 6 is a component of *Arm® DS-5 Development Studio* and *Arm® Keil® MDK*. Alternatively, you can use Arm Compiler 6 as a *standalone product*. The features and processors that Arm Compiler 6 supports depend on the product edition. See *Arm® Developer* for the specification of the different standard products.

Arm Compiler 6 combines the optimized tools and libraries from Arm with a modern LLVM-based compiler framework. The components in Arm Compiler 6 are:

armclang
: The compiler and integrated assembler that compiles C, C++, and GNU assembly language sources.

  The compiler is based on LLVM and Clang technology.

  Clang is a compiler front end for LLVM that supports the C and C++ programming languages.

armasm
: The legacy assembler. Only use `armasm` for legacy Arm-syntax assembly code. Use the `armclang` assembler and GNU syntax for all new assembly files.

armlink
: The linker 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 archiver 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 application developer.

fromelf
: The image conversion utility can convert Arm ELF images to binary formats and can also generate textual information about the input image, such as its disassembly and its code and data size.

Arm C++ libraries
: The Arm C++ libraries are based on the LLVM libc++ project:
  - The libc++abi library is a runtime library providing implementations of low-level language features.
  - The libc++ library provides an implementation of the ISO C++ library standard. It depends on the functions that are 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.

Application development

A typical application development flow 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.

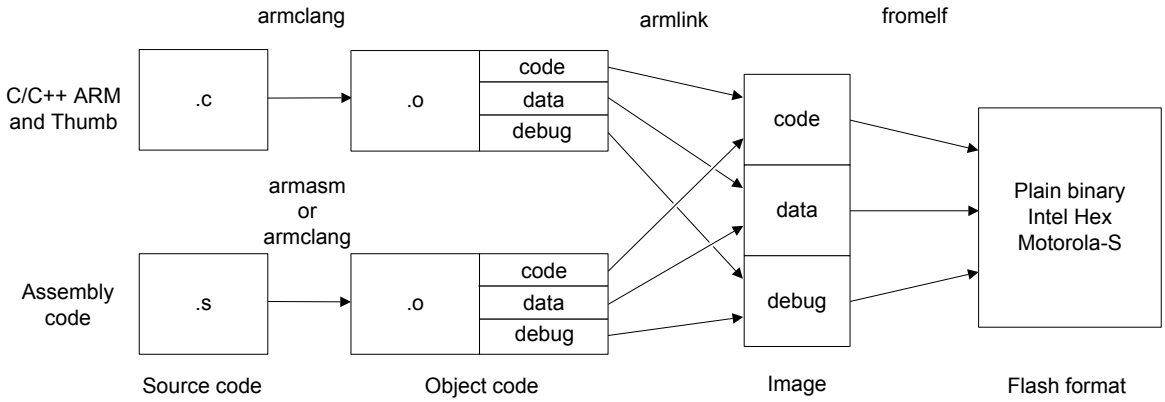


Figure 1-1 A typical tool usage flow diagram

Arm Compiler 6 has more functionality than the set of product features that are described in the documentation. The various features in Arm Compiler 6 can have different levels of support and guarantees. For more information, see *Support level definitions* on page Appx-A-127.

Note

If you are migrating your toolchain from Arm Compiler 5 to Arm Compiler 6, then see the *Arm® Compiler Migration and Compatibility Guide* for information on how to migrate your source code and toolchain build options from Arm Compiler 5 to Arm Compiler 6. For a list of *Arm® Compiler 6 documents*, see the documentation on Arm Developer.

1.2 Installing Arm® Compiler

This topic lists the system requirements for running Arm Compiler, and then guides you through the installation process.

System Requirements

Arm Compiler 6 is available for the following operating systems:

- Windows 64-bit.
- Windows 32-bit.
- Linux 64-bit.

For more information on system requirements see the *Arm® Compiler release note*.

Installing Arm® Compiler

You can install Arm Compiler as a standalone product on supported Windows and Linux platforms. If you use Arm Compiler as part of a development suite such as Arm DS-5 Development Studio or Arm Keil µVision® IDE, then installing the development suite also installs Arm Compiler. The following instructions are for installing Arm Compiler as a standalone product.

Prerequisites:

1. *Download Arm® Compiler 6*.
2. Obtain a license. Contact your Arm sales representative or *request a license*.
3. Set the `ARMLMD_LICENSE_FILE` environment variable to point to your license file or license server.

   Note

   This path must not contain double quotes on Windows. A path that contains spaces still works without the quotes.

If you need to set any other environment variable, such as `ARM_TOOL_VARIANT`, see *Toolchain environment variables* on page Appx-A-134 for more information.

Installing a standalone Arm® Compiler on Windows platforms

To install Arm Compiler as a standalone product on Windows, you need the `setup.exe` installer on your machine. This is in the *Arm® Compiler 6 download*:

1. On 64-bit platforms, run `win-x86_64\setup.exe`. On 32-bit platforms, run `win-x86_32\setup.exe`.
2. Follow the on-screen installation instructions.

If you have an older version of Arm Compiler 6 and you want to upgrade, Arm recommends that you uninstall the older version of Arm Compiler 6 before installing the new version of Arm Compiler 6.

Installing a standalone Arm® Compiler on Linux platforms

To install Arm Compiler as a standalone product on Linux platforms, you need the `install_x86_64.sh` installer on your machine. This is in the *Arm® Compiler 6 download*:

1. Run `install_x86_64.sh` normally, without using the `source` Linux command.
2. Follow the on-screen installation instructions.

Uninstalling a standalone Arm® Compiler

To uninstall Arm Compiler on Windows, use the Control Panel:

1. Select **Control Panel** > **Programs and Features**.
2. Select the version that you want to uninstall, for example **ARM Compiler 6.5**.
3. Click the **Uninstall** button.

To uninstall Arm Compiler on Linux, delete the Arm Compiler 6 installation directory for the compiler version you want to delete.

For more information on installation, see the *Arm® Compiler release note*.

Related tasks

1.3 Accessing Arm® Compiler from Arm® DS-5 Development Studio on page 1-17.
1.4 Accessing Arm® Keil® µVision® IDE on page 1-19.

1.3 Accessing Arm® Compiler from Arm® DS-5 Development Studio

Arm DS-5 is a development suite that provides Arm Compiler 6 as a built-in toolchain.

This task describes how to access and configure Arm Compiler from the DS-5 environment.

Prerequisites

Ensure you have DS-5 installed. Create a new C or C++ project in DS-5. For information on creating new projects in DS-5, see *Creating a new C or C++ project*.

Procedure

1. Select the project in DS-5.
2. Select **Project** > **Properties**.
3. From the left-hand side menu, select **C/C++ Build** > **Tool Chain Editor**.
4. In the `Current toolchain` options, select **ARM Compiler 6** if this is not already selected.
5. From the left-hand side menu, select **C/C++ Build** > **Settings**.

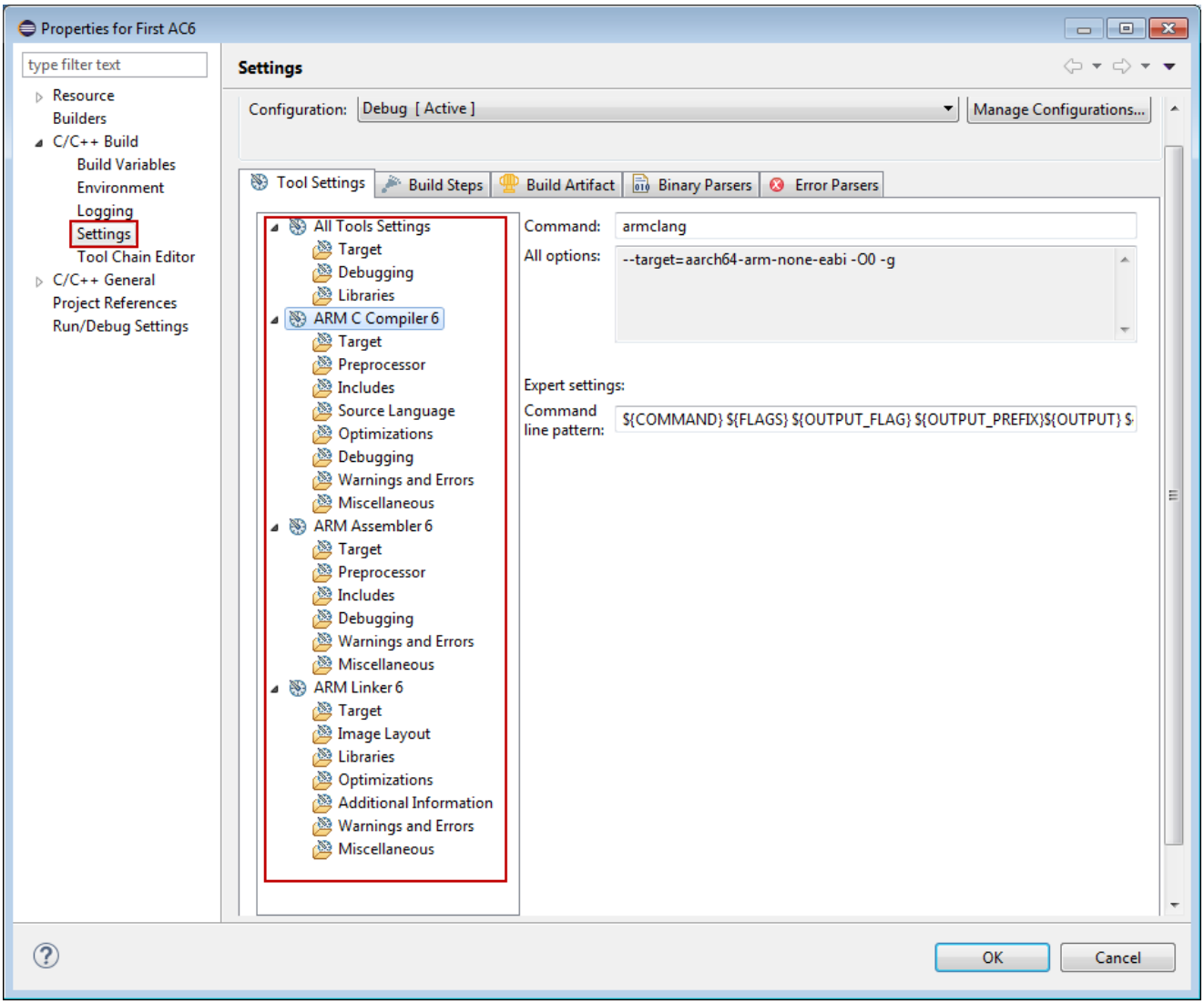


Figure 1-2 Accessing Arm Compiler settings from DS-5

For information about using DS-5, see the *Arm® DS-5 Getting Started Guide* and *Arm® DS-5 Debugger Guide*.

6. After setting the compiler options right-click on the project and select **Build Project**.

Related references

1.2 Installing Arm® Compiler on page 1-15.

1.4 Accessing Arm® Compiler from Arm® Keil® µVision® IDE

Arm Keil µVision IDE is a microprocessor development suite that provides Arm Compiler 6 as a built-in toolchain.

This task describes how to access and configure Arm Compiler from the Keil µVision environment:

Prerequisites

Ensure you have Keil µVision installed. Create a new project in µVision.

Procedure

1. Select the project in µVision.
2. Select **Project** > **Manage** > **Project '*project_name*' Project Items**.

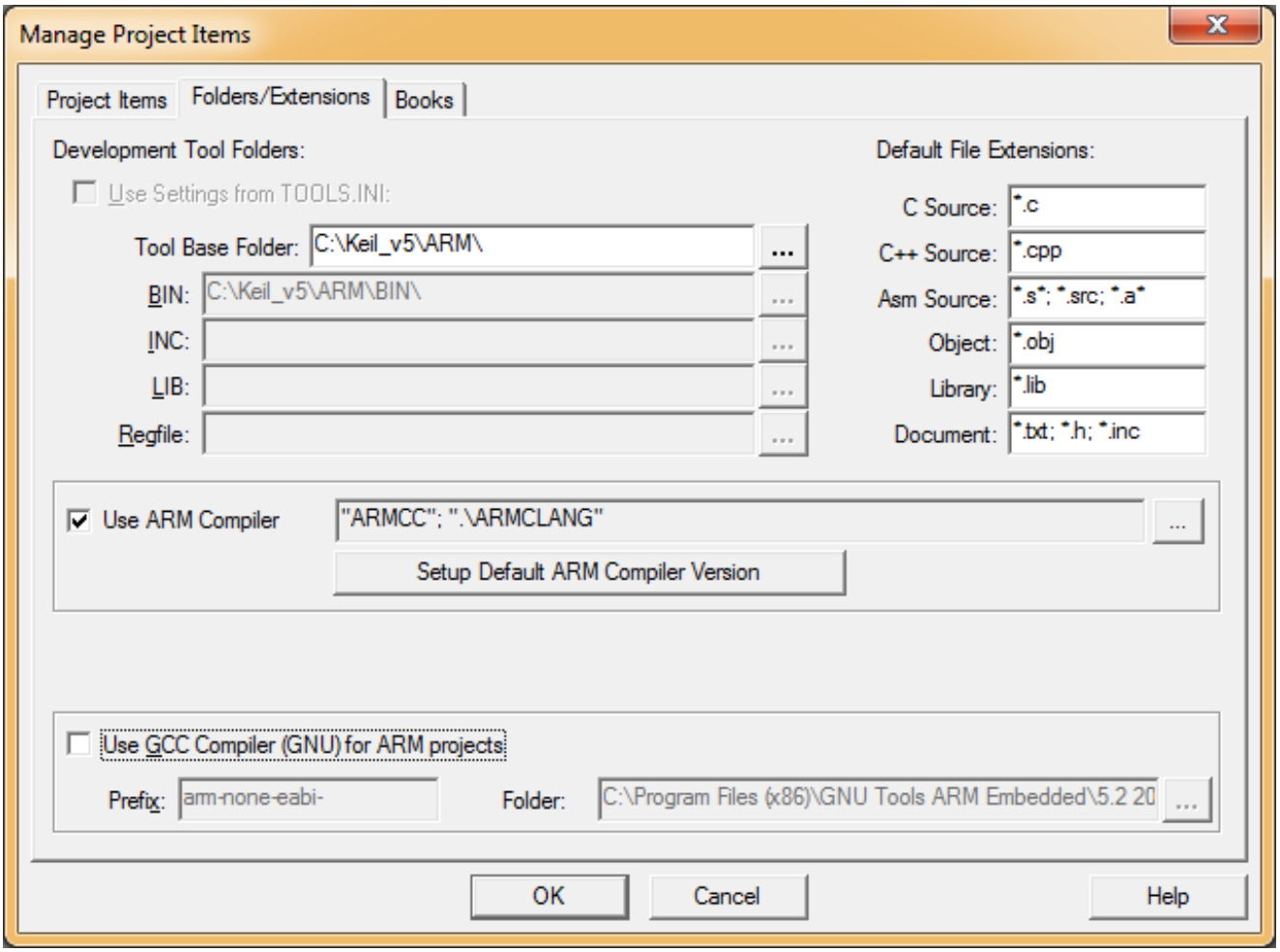


Figure 1-3 Accessing the Arm Compiler version from Keil µVision

3. Select the **Folders/Extensions** tab.
4. Click **Setup Default ARM Compiler Version**.
5. For each device, select the version of Arm Compiler you want to use. For example, **v6.9**.
6. Click **OK** to close each of the dialog boxes in turn.

Related references

1.2 Installing Arm® Compiler on page 1-15.

1.5 Compiling a Hello World example

These examples show how to use the Arm Compiler toolchain to build and inspect an executable image from C/C++ source files.

The source code

The source code that is used in the examples is a single C source file, `hello.c`, to display a greeting message:

```
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

Compiling in a single step

When compiling code, you must first decide which target the executable is to run on. An Armv8-A target can run in different states:

- AArch64 state targets execute A64 instructions using 64-bit and 32-bit general-purpose registers.
- AArch32 state targets execute A32 or T32 instructions using 32-bit general-purpose registers.

The `--target` option determines which target state to compile for. This option is a mandatory option.

Compiling for an AArch64 target

To create an executable for an AArch64 target in a single step:

```
armclang --target=aarch64-arm-none-eabi hello.c
```

This command creates an executable, `a.out`.

This example compiles for an AArch64 state target. Because only `--target` is specified, the compiler defaults to generating code that runs on any Armv8-A target. You can also use `-mcpu` to target a specific processor.

Compiling for an AArch32 target

To create an executable for an AArch32 target in a single step:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 hello.c
```

There is no default target for AArch32 state. You must specify either `-march` to target an architecture or `-mcpu` to target a processor. This example uses `-mcpu` to target the Cortex-A53 processor. The compiler generates code that is optimized specifically for the Cortex-A53, but might not run on other processors.

Use `-mcpu=list` or `-march=list` to see all available processor or architecture options.

Beyond the defaults

Compiler options let you specify precisely how the compiler behaves when generating code.

The *armclang Reference Guide* describes all the supported options, but here are some of the most common:

- Including debug information. The `-g` option tells the compiler to produce DWARF debug information. You can then use a compatible debugger, such as Arm DS-5 Debugger, to load, run, and debug images.
- Optimization. The `-O`*`level`* option specifies the level of optimization to use when compiling source files. The default is `-O0`, with no optimization. Different optimization levels let you control what type of optimization the compiler performs. For example, `-Os` aims to reduce code size by balancing code

size against code speed, whereas `-Omax` uses aggressive optimizations to target performance optimization.
- Instruction set. AArch32 targets support two instruction sets that you specify with the `-m` option. The `-marm` option specifies A32, that is 32-bit instructions, to emphasize performance. The `-mthumb` option specifies T32, that is mixed 32-bit and 16-bit instructions, to emphasize code density.

Examining the executable

The `fromelf` tool lets you examine a compiled binary, extract information about it, or convert it.

For example, you can:
- Disassemble the code that is contained in the executable:

```
fromelf --text -c a.out

    ...
    main
    0x000081a0:    e92d4800    .H-.    PUSH     {r11,lr}
    0x000081a4:    e1a0b00d    ....    MOV      r11,sp
    0x000081a8:    e24dd010    ..M.    SUB      sp,sp,#0x10
    0x000081ac:    e3a00000    ....    MOV      r0,#0
    0x000081b0:    e50b0004    ....    STR      r0,[r11,#-4]
    0x000081b4:    e30a19cc    ....    MOV      r1,#0xa9cc
    ...
```

- Examine the size of code and data in the executable:

```
fromelf --text -z a.out

    Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name
    10436        492        596         16        348       3468   a.out
    10436        492        596         16          0          0   ROM Totals for a.out
```

- Convert the ELF executable image to another format, for example a plain binary file:

```
fromelf --bin --output=outfile.bin a.out
```

See *fromelf Command-line Options* for the options from the `fromelf` tool.

Compiling and linking as separate steps

For simple projects with small numbers of source files, compiling and linking in a single step might be the simplest option:

```
armclang --target=aarch64-arm-none-eabi file1.c file2.c -o image.axf
```

This example compiles the two source files `file1.c` and `file2.c` for an AArch64 state target. The `-o` option specifies that the filename of the generated executable is `image.axf`.

More complex projects might have many more source files. It is not efficient to compile every source file at every compilation, because most source files are unlikely to change. To avoid compiling unchanged source files, you can compile and link as separate steps. In this way, you can then use a build system (such as `make`) to compile only those source files that have changed, then link the object code together. The `armclang -c` option tells the compiler to compile to object code and stop before calling the linker:

```
armclang -c --target=aarch64-arm-none-eabi file1.c
armclang -c --target=aarch64-arm-none-eabi file2.c
armlink file1.o file2.o -o image.axf
```

These commands do the following:
- Compile `file1.c` to object code, and save using the default name `file1.o`.
- Compile `file2.c` to object code, and save using the default name `file2.o`.
- Link the object files `file1.o` and `file2.o` to produce an executable that is called `image.axf`.

In the future, if you modify `file2.c`, you can rebuild the executable by recompiling only `file2.c` then linking the new `file2.o` with the existing `file1.o` to produce a new executable:

```
armclang -c --target=aarch64-arm-none-eabi file2.c
armlink file1.o file2.o -o image.axf
```

Related information

armclang --target option.
armclang -march option.
armclang -mcpu option.
Summary of armclang command-line options.

1.6 Using the integrated assembler

These examples show how to use the `armclang` integrated assembler to build an object from assembly source files, and how to call functions in this object from C/C++ source files.

The assembly source code

The assembly example is a single assembly source file, `mystrcopy.s`, containing a function to perform a simple string copy operation:

```
    .section   StringCopy, "ax"
    .balign    4

    .global    mystrcopy
    .type      mystrcopy, "function"
mystrcopy:
    ldrb       r2, [r1], #1
    strb       r2, [r0], #1
    cmp        r2, #0
    bne        mystrcopy
    bx         lr
```

The `.section` directive creates a new section in the object file named `StringCopy`. The characters in the string following the section name are the *flags* for this section. The `a` flag marks this section as allocatable. The `x` flag marks this section as executable.

The `.balign` directive aligns the subsequent code to a 4-byte boundary. The alignment is required for compliance with the *Arm® Application Procedure Call Standard* (AAPCS).

The `.global` directive marks the symbol `mystrcopy` as a global symbol. This enables the symbol to be referenced by external files.

The `.type` directive sets the type of the symbol `mystrcopy` to `function`. This helps the linker use the proper linkage when the symbol is branched to from A32 or T32 code.

Assembling a source file

When assembling code, you must first decide which target the executable is to run on. The `--target` option determines which target state to compile for. This option is a mandatory option.

To assemble the above source file for an Armv8-M Mainline target:

```
armclang --target=arm-arm-none-eabi -c -march=armv8-m.main mystrcopy.s
```

This command creates an object file, `mystrcopy.o`.

The `--target` option selects the target that you want to assemble for. In this example, there is no default target for A32 state, so you must specify either `-march` to target an architecture or `-mcpu` to target a processor. This example uses `-march` to target the Armv8-M Mainline architecture. The integrated assembler accepts the same options for `--target`, `-march`, `-mcpu`, and `-mfpu` as the compiler.

Use `-mcpu=list` or `-march=list` to see all available options.

Examining the executable

You can use the `fromelf` tool to:

- examine an assembled binary.
- extract information about an assembled binary.
- convert an assembled binary to another format.

For example, you can disassemble the code that is contained in the object file:

```
fromelf --text -c mystrcopy.o

    ...
    ** Section #3 'StringCopy' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
        Size   : 14 bytes (alignment 4)
        Address: 0x00000000
```

```
        $t.0
        mystrcopy
            0x00000000:    f8112b01    ...+    LDRB     r2,[r1],#1
            0x00000004:    f8002b01    ...+    STRB     r2,[r0],#1
            0x00000008:    2a00        .*      CMP      r2,#0
            0x0000000a:    d1f9        ..      BNE      mystrcopy ; 0x0
            0x0000000c:    4770        pG      BX       lr
    ...
```

The example shows the disassembly for the section `StringCopy` as created in the source file.

Note

The code is marked as T32 by default because Armv8-M Mainline does not support A32 code. For processors that support A32 and T32 code, you can explicitly mark the code as A32 or T32 by adding the GNU assembly `.arm` or `.thumb` directive, respectively, at the start of the source file.

Calling an assembly function from C/C++ code

It can be useful to write optimized functions in an assembly file and call them from C/C++ code. When doing so, ensure that the assembly function uses registers in compliance with the AAPCS.

The C example is a single C source file `main.c`, containing a call to the `mystrcopy` function to copy a string from one location to another:

```
const char *source = "String to copy.";
char *dest;

extern void mystrcopy(char *dest, const char *source);

int main(void) {
    mystrcopy(dest, source);
    return 0;
}
```

An `extern` function declaration has been added for the `mystrcopy` function. The return type and function parameters must be checked manually.

If you want to call the assembly function from a C++ source file, you must disable C++ name mangling by using `extern "C"` instead of `extern`. For the above example, use:

```
extern "C" void mystrcopy(char *dest, const char *source);
```

Compiling and linking the C source file

To compile the above source file for an Armv8-M Mainline target:

```
armclang --target=arm-arm-none-eabi -c -march=armv8-m.main main.c
```

This command creates an object file, `main.o`.

To link the two object files `main.o` and `mystrcopy.o` and generate an executable image:

```
armlink main.o mystrcopy.o -o image.axf
```

This command creates an executable image file `image.axf`.

Related concepts

2.1 Mandatory armclang options on page 2-29.

Related information

Summary of armclang command-line options.

Sections.

1.7 Running bare-metal images

By default, Arm Compiler produces bare-metal images. Bare-metal images can run without an operating system. The images can run on a hardware target or on a software application that simulates the target, such as Fast Models or Fixed Virtual Platforms.

If you are using Arm DS-5, you can select **Run** > **Debug Configurations** to configure and load your application image into either a model or hardware platform.

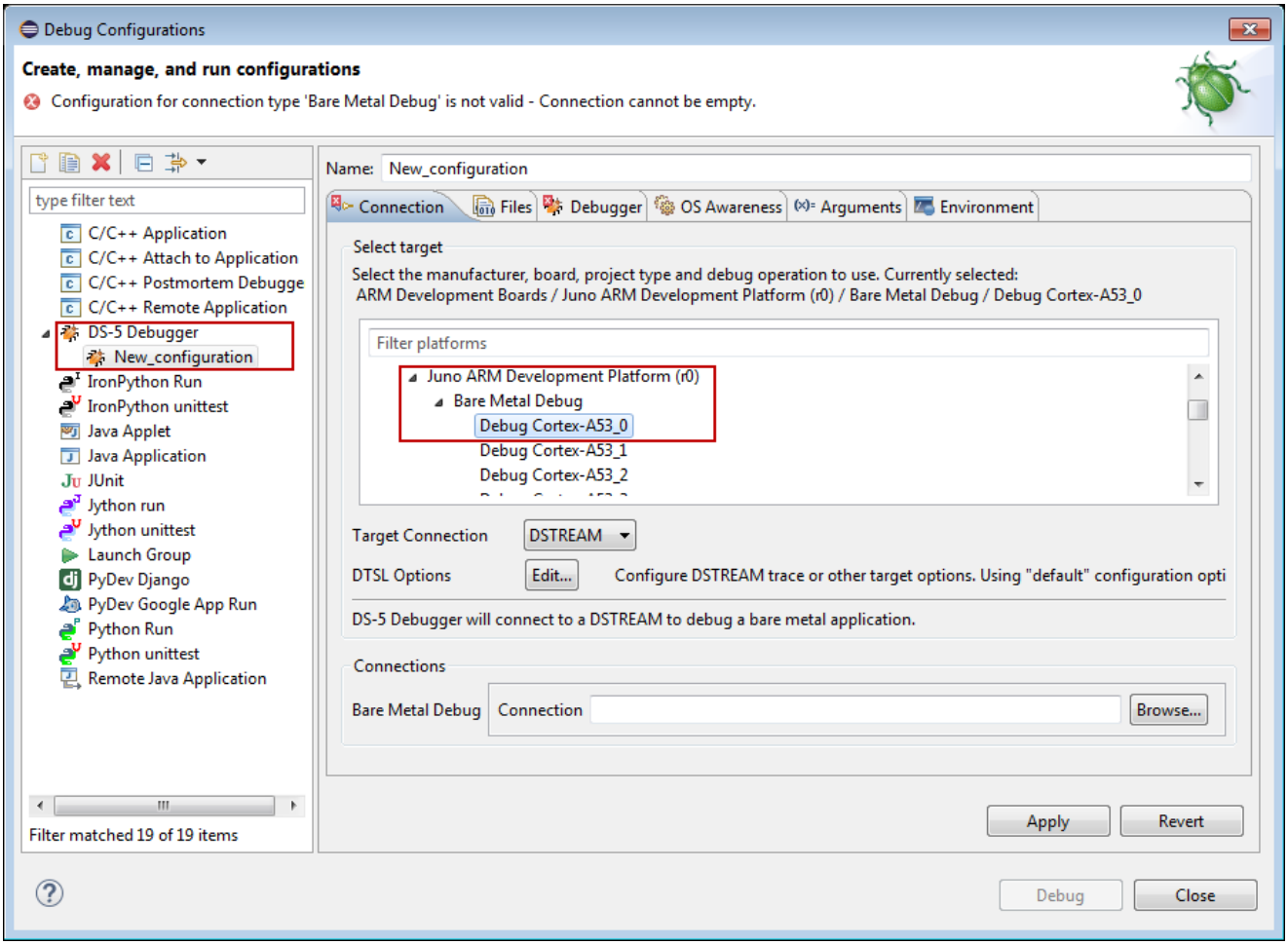


Figure 1-4 Debug configurations

For more information on configuring and running the image using Arm DS-5, see the *Arm® DS-5 Debugger User Guide*.

By default, the C library in Arm Compiler uses special functions to access the input and output interfaces on the host computer. These functions implement a feature called semihosting. Semihosting is useful when the input and output on the hardware is not available during the early stages of application development.

When you want your application to use the input and output interfaces on the hardware, you must retarget the required semihosting functions in the C library.

For more information on configuring the Arm DS-5 Debugger settings, see *Configuring a connection to a bare-metal hardware target*.

Outputting debug messages from your application

The semihosting feature enables your bare-metal application, running on an Arm processor, to use the input and output interface on a host computer. This feature requires the use of a debugger that supports semihosting, for example Arm DS-5 Debugger, on the host computer.

A bare-metal application that uses semihosting does not use the input and output interface of the development platform. When the input and output interfaces on the development platform are available, you must reimplement the necessary semihosting functions to use the input and output interfaces on the development platform.

For more information, see how to use the libraries in *semihosting* and *nonsemihosting* environments.

Related information

Arm DS-5 Debugger User Guide.

1.8 Architectures supported by Arm® Compiler

Arm Compiler supports a number of different architecture profiles.

Arm Compiler supports the following architectures:

- Armv8-A and all update releases, for bare-metal targets.
- Armv8-R.
- Armv8-M.
- Armv7-A for bare-metal targets.
- Armv7-R.
- Armv7-M.
- Armv6-M.

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, the Cortex-A53 processor).

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 `-march` or `-mcpu` is specified.

`arm-arm-none-eabi`
: Generates A32 and 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`, and `-mcpu` options are `armclang` options. For all of the other tools, such as `armasm` and `armlink`, use the `--cpu` option to specify target processors and architectures.

Related information

armclang --target option.
armclang -march option.
armclang -mcpu option.
armlink --cpu option.
Arm Glossary.

Chapter 2
Using Common Compiler Options

There are many options that you can use to control how Arm Compiler 6 generates code for your application. This section lists the mandatory and commonly used optional command-line arguments, such as to control target selection, optimization, and debug view.

It contains the following sections:

2.1 Mandatory armclang options

When using `armclang`, you must specify a target on the command-line. Depending on the target you use, you might also have to specify an architecture or processor.

Specifying a target

To specify a target, use the *`--target`* option. The following targets are available:

- To generate A64 instructions for AArch64 state, specify `--target=aarch64-arm-none-eabi`.

  Note

  For AArch64, the default architecture is Armv8-A.

- To generate A32 and T32 instructions for AArch32 state, specify `--target=arm-arm-none-eabi`. To specify generation of either A32 or T32 instructions, use *`-marm`* or *`-mthumb`* respectively.

  Note

  AArch32 has no defaults. You must always specify an architecture or processor.

Specifying an architecture

To generate code for a specific architecture, use the *`-march`* option. The supported architectures vary according to the selected target.

To see a list of all the supported architectures for the selected target, use `-march=list`.

Specifying a processor

To generate code for a specific processor, use the *`-mcpu`* option. The supported processors vary according to the selected target.

To see a list of all the supported processors for the selected target, use `-mcpu=list`.

It is also possible to enable or disable optional architecture features, by using the `+[no]feature` notation. For a list of the architecture features that your processor supports, see the processor product documentation. See the *armclang Reference Guide* for a *list of architecture features* that Arm Compiler supports.

Use *`+feature`* or `+no`*`feature`* to explicitly enable or disable an optional architecture feature.

Note

You do not need to specify both the architecture and processor. The compiler infers the architecture from the processor. If you only want to run code on one particular processor, you can specify the specific processor. Performance is optimized, but code is only guaranteed to run on that processor. If you want your code to run on a range of processors from a particular architecture, you can specify the architecture. The code runs on any processor implementation of the target architecture, but performance might be impacted.

Examples

These examples compile and link the input file `helloworld.c`:

- To compile for the Armv8-A architecture in AArch64 state, use:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a helloworld.c
```

- To compile for the Armv8-R architecture in AArch32 state, use:

```
armclang --target=arm-arm-none-eabi -march=armv8-r helloworld.c
```

- To compile for the Armv8-M architecture mainline profile, use:

```
armclang --target=arm-arm-none-eabi -march=armv8-m.main helloworld.c
```

- To compile for a Cortex-A53 processor in AArch64 state, use:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 helloworld.c
```

- To compile for a Cortex-A53 processor in AArch32 state, use:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 helloworld.c
```

- To compile for a Cortex-M4 processor, use:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m4 helloworld.c
```

- To compile for a Cortex-M33 processor, with DSP disabled, use:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m33+nodsp helloworld.c
```

Related information

armclang --target option.
armclang -march option.
armclang -mcpu option.
armclang -marm option.
armclang -mthumb option.
Summary of armclang command-line options.

2.2 Selecting source language options

Arm Compiler infers the source language, for example C or C++, from the filename extension. You can force Arm Compiler to compile for a specific source language using the `-x` option. Arm Compiler can also compile different variants of C and C++ source code. You can specify the variant using the `-std` option.

Note

This topic includes descriptions of [BETA] and [COMMUNITY] features. See *Support level definitions on page Appx-A-127*.

Source language

By default Arm Compiler treats files with `.c` extension as C source files. If you want to compile a `.c` file, for example `file.c`, as a C++ source file, use the `-xc++` option:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -xc++ file.c
```

By default Arm Compiler treats files with `.cpp` extension as C++ source files. If you want to compile a `.cpp` file, for example `file.cpp`, as a C source file, use the `-xc` option:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -xc file.cpp
```

The `-x` option only applies to input files that follow it on the command line.

Source language standard

Arm Compiler supports Standard and GNU variants of source languages as shown in the following table.

Table 2-1 Source language variants

Standard C	GNU C	Standard C++	GNU C++
c90	gnu90	c++98	gnu++98
c99	gnu99	c++03	-
c11 [COMMUNITY]	gnu11	c++11	gnu++11
-	-	c++14 [BETA]	gnu++14 [BETA]

The default language standard for C code is `gnu11`. The default language standard for C++ code is gnu++98. To specify a different source language standard, use the `-std=`*`name`* option.

Arm Compiler supports various language extensions, including GCC extensions, which you can use in your source code. The GCC extensions are only available when you specify one of the GCC C or C++ language variants. For more information on language extensions, see the *Arm® C Language Extensions* in Arm Compiler.

Since Arm Compiler uses the available language extensions by default, it does not adhere to the strict ISO Standard. To compile to strict ISO standard for the source language, use the `-Wpedantic` option. This shows warnings where the source code violates the ISO Standard. Arm Compiler does not support strict adherence to C++98 or C++03.

If you do not use `-Wpedantic`, Arm Compiler uses the available language extensions without warning. However, where language variants produce different behavior, the behavior of the language variant specified by `-std` will apply.

Note

Certain compiler optimizations can violate strict adherence to the ISO Standard for the language. To identify when these violations happen, use the `-Wpedantic` option.

The following example shows the use of a variable length array, which is a C99 feature. In this example, the function declares an array `i`, with variable length *n*.

```
#include <stdlib.h>

void function(int n) {
    int i[n];
}
```

Arm Compiler does not warn when compiling the example for C99 with `-Wpedantic`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c -std=c99 -Wpedantic file.c
```

Arm Compiler does warn about variable length arrays when compiling the example for C90 with `-Wpedantic`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c -std=c90 -Wpedantic file.c
```

In this case `armclang` gives the following warning:

```
file.c:4:8: warning: variable length arrays are a C99 feature [-Wvla-extension]
int i[n];
^
1 warning generated.
```

Related information

Standard C++ library implementation definition.

2.3 Selecting optimization options

Arm Compiler performs several optimizations to reduce the code size and improve the performance of your application. However, optimization techniques can result in the loss of debug information, increased build time, or increase in image size. Optimization levels are always a trade-off between these three parameters.

Arm Compiler provides optimization options for the different optimization trade-offs. Primarily, you can optimize for performance or for image size. However, there are several options for finer control of the optimizations techniques. The optimization options are:

`-O0`
: This is the default optimization setting. It turns off most optimizations, and gives the best correlation between the built image and your application source code.

Use the following options to optimize performance:

`-O1`
: This results in more optimizations for performance, when compared to `-O0`. It also reduces the information available for debugging, and might result in an increased image size. Arm recommends this option for debugging.

`-O2`
: This results in more optimizations for performance, when compared to `-O1`. It also reduces the information available for debugging, and might result in an increased image size.

`-O3`
: This results in more optimizations for performance, when compared to `-O2`. It also reduces the information available for debugging, and might result in an increased image size.

`-Ofast`
: This results in more optimizations for performance, when compared to `-O3`. It also reduces the information available for debugging, and might result in an increased image size. At this optimization level, Arm Compiler might violate certain language standards.

`-Omax`
: This results in more optimizations for performance, when compared to `-Ofast`. It also reduces the information available for debugging, and might result in an increased image size. At this optimization level, Arm Compiler might violate certain language standards. Arm recommends this option for best performance.

Use the following options to optimize for size:

`-Os`
: This results in reduced code size, and also reduces the information available for debugging. Using this option might make your code slower.

`-Oz`
: This results in more reduced image size, when compared to `-Os`, and also reduces the information available for debugging. Using this option is likely to make your code slower than `-Os`. Arm recommends this option for best code size.

The example shows the optimization performed with the `-O1` optimization option. To perform this optimization, compile your source file using:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O1 -c -S file.c
```

Table 2-2 Optimization example

Source code in file.c	Optimized output from `armclang`

`int dummy()`
`{`
`  int x=10, y=20;`
`  int z;`
`  z=x+y;`
`  return 0;`
`}`
	`dummy:`
	`  .fnstart`
	`  movs r0, #0`
	`  bx lr`

The example shows the optimization performed with the -O0 optimization option. To perform this optimization, compile your source file using:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O0 -c -S file.c
```

Table 2-3 Optimization example

Source code in file.c	Unoptimized output from `armclang`

`int dummy()`
`{`
`  int x=10, y=20;`
`  int z;`
`  z=x+y;`
`  return 0;`
`}`
	`dummy:`
	`  .fnstart`
	`  .pad #12`
	`  sub  sp, sp, #12`
	`  mov  r0, #10`
	`  str  r0, [sp, #8]`
	`  mov  r0, #20`
	`  str  r0, [sp, #4]`
	`  ldr  r0, [sp, #8]`
	`  add  r0, r0, #20`
	`  str  r0, [sp]`
	`  mov  r0, #0`
	`  add  sp, sp, #12`
	`  bx  lr`

2.4 Building to aid debugging

During application development, you must debug the image that you build. The Arm Compiler tools have various features that provide good debug view and enable source-level debugging, such as setting breakpoints in C and C++ code. There are also some features you must avoid when building an image for debugging.

Available command-line options

To build an image for debugging, you must compile with the `-g` option. This option allows you to specify the DWARF format to use. The `-g` option is a synonym for `-gdwarf-4`. You can specify DWARF 2 or DWARF 3 if necessary, for example:

```
armclang -gdwarf-3
```

When linking, there are several `armlink` options available to help improve the debug view:

- `--debug`. This option is the default.
- `--no_remove` to retain all input sections in the final image even if they are unused.
- `--bestdebug`. When different input objects are compiled with different optimization levels, this option enables linking for the best debug illusion.

Effect of optimizations on the debug view

To build an application that gives the best debug view, it is better to use options that give the fewest optimizations. Arm recommends using optimization level `-O1` for debugging. This option gives good code density with a satisfactory debug view.

Higher optimization levels perform progressively more optimizations with correspondingly poorer debug views.

The compiler attempts to automatically inline functions at optimization levels `-O2` and `-O3`. If you must use these optimization levels, disable the automatic inlining with the `armclang` option `-fno-inline-functions`. The linker inlining is disabled by default.

Support for debugging overlaid programs

The linker provides various options to support overlay-aware debuggers:

- `--emit_debug_overlay_section`
- `--emit_debug_overlay_relocs`

These options permit an overlay-aware debugger to track which overlay is active.

Features to avoid when building an image for debugging

Avoid using the following in your source code:

- The `__attribute__((always_inline))` function attribute. Qualifying a function with this attribute forces the compiler to inline the function. If you also use the `-fno-inline-functions` option, the function is inlined.
- The `__declspec(noreturn)` attribute and the `__attribute__((noreturn))` function attribute. These attributes limit the ability of a debugger to display the call stack.

Avoid using the following features when building an image for debugging:

- Link time optimization. This feature performs aggressive optimizations and can remove large chunks of code.
- The `armlink --no_debug` option.
- The `armlink --inline` option. This option changes the image in such a way that the debug information might not correspond to the source code.

2.5 Linker options for mapping code and data to target memory

For an image to run correctly on a target, you must place the various parts of the image at the correct locations in memory. Linker command-line options are available to map the various parts of an image to target memory.

The options implement the scatter-loading mechanism that describes the memory layout for the image. The options that you use depend on the complexity of your image:

- For simple images, use the following memory map related options:
  - `--ro_base` to specify the address of both the load and execution region containing the RO output section.
  - `--rw_base` to specify the address of the execution region containing the RW output section.
  - `--zi_base` to specify the address of the execution region containing the ZI output section.

  Note

  For objects that include *execute-only* (XO) sections, the linker provides the `--xo_base` option to locate the XO sections. These sections are objects that are targeted at Armv7-M or Armv8-M architectures, or objects that are built with the `armclang -mthumb` option,

- For complex images, use a text format scatter-loading description file. This file is known as a scatter file, and you specify it with the `--scatter` option.

Note

You cannot use the memory map related options with the `--scatter` option.

Examples

The following example shows how to place code and data using the memory map related options:

```
armlink --ro_base=0x0 --rw_base=0x400000 --zi_base=0x405000 --first="init.o(init)" init.o
main.o
```

Note

In this example, `--first` is also included to make sure that the initialization routine is executed first.

The following example shows a scatter file, `scatter.scat`, that defines an equivalent memory map:

```
LR1 0x0000 0x20000
{
    ER_RO 0x0
    {
        init.o (INIT, +FIRST)
        *(+RO)
    }

    ER_RW 0x400000
    {
        *(+RW)
    }

    ER_ZI 0x405000
    {
        *(+ZI)
    }
}
```

To link with this scatter file, use the following command:

```
armlink --scatter=scatter.scat init.o main.o
```

2.6 Controlling diagnostic messages

Arm Compiler provides diagnostics messages in the form of warnings and errors. You can use options to suppress these messages or enable them as either warnings or errors.

Arm Compiler lists all the warnings and errors it encounters during the compiling and linking process. However, if you specify multiple source files, and Arm Compiler encounters an error from a source file, it does not report any diagnostic information from the other source files that it has not processed.

Diagnostic messages from Arm Compiler include the following information:

- Name of file that contains the error or warning.
- Line number in the file that contains the error or warning.
- Character in the line that is associated with the error or warning.
- Description of the error or warning.
- A diagnostic flag of the form `-W`*`flag`*, for example `-Wvla-extension`, to identify the error or warning. Only the messages that you can suppress have an associated flag. Errors that you cannot suppress do not have an associated flag.

An example warning diagnostic message is:

```
file.c:8:7: warning: variable length arrays are a C99 feature [-Wvla-extension]
 int i[n];
      ^
```

This warning message tells you:

- The file that contains the problem is called `file.c`.
- The problem is on line `8` of `file.c`, and starts at character `7`.
- The warning is about the use of a variable length array `i[n]`.
- The flag to identify, enable or disable this diagnostic message is `vla-extension`.

The following are common options that control diagnostic output from Arm Compiler.

Table 2-4 Common diagnostic options

Option	Description
`-Werror`	Turn all warnings into errors.
`-Werror=foo`	Turn warning flag `foo` into an error.
`-Wno-error=foo`	Leave warning flag `foo` as a warning even if `-Werror` is specified.
`-Wfoo`	Enable warning flag `foo`.
`-Wno-foo`	Suppress warning flag `foo`.
`-w`	Suppress all warnings. Note that this is a lowercase `w`.
`-Weverything`	Enable all warnings.

See *Controlling Errors and Warnings* in the *Clang Compiler User's Manual* for full details about controlling diagnostics with `armclang`.

Examples of controlling diagnostic messages

Copy the following code example to `file.c` and compile it with Arm Compiler to see example diagnostic messages.

```
#include <stdlib.h>
#include <stdio.h>

void function (int x) {
    int i;
    int y=i+x;
```

```
    printf("Result of %d plus %d is %d\n", i, x); /* Missing an input argument for the third
%d */
    call(); /* This function has not been declared and is therefore an implicit declaration
*/

    return;
}
```

Compile `file.c` using:

```
armclang --target=aarch64-arm-none-eabi -march=armv8 -c file.c
```

By default `armclang` checks the format of `printf()` statements to ensure that the number of `%` format specifiers matches the number of data arguments. Therefore Arm Compiler generates this diagnostic message:

```
file.c:9:36: warning: more '%' conversions than data arguments [-Wformat]
                printf("Result of %d plus %d is %d\n", i, x);
                                                   ^
```

By default `armclang` compiles for the `gnu11` standard for `.c` files. This language standard does not allow implicit function declarations. Therefore Arm Compiler generates this diagnostic message:

```
file.c:11:3: warning: implicit declaration of function 'call' is invalid C99 [-Wimplicit-
function-declaration]
    call();
    ^
```

To suppress all warnings, use `-w`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c file.c -w
```

To suppress only the `-Wformat` warning, use `-Wno-format`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c file.c -Wno-format
```

To enable the `-Wformat` message as an error, use `-Werror=format`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c file.c -Werror=format
```

Some diagnostic messages are suppressed by default. To see all diagnostic messages use `-Weverything`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c file.c -Weverything
```

2.7 Selecting floating-point options

Arm Compiler supports floating-point arithmetic and floating-point data types in your source code or application. Arm Compiler supports floating-point arithmetic either by using libraries that implement floating-point arithmetic in software, or by using the hardware floating-point registers and instructions that are available on most Arm-based processors.

You can use various options that determine how Arm Compiler generates code for floating-point arithmetic. Depending on your target, you might need to specify one or more of these options to generate floating-point code that correctly uses floating-point hardware or software libraries.

Table 2-5 Options for floating-point selection

Option	Description
`armclang -mfpu`	Specify the floating point architecture to the compiler.
`armclang -mfloat-abi`	Specify the floating-point linkage to the compiler.
`armclang -march`	Specify the target architecture to the compiler. This automatically selects the default floating-point architecture.
`armclang -mcpu`	Specify the target processor to the compiler. This automatically selects the default floating-point architecture.
`armlink --fpu`	Specify the floating-point architecture to the linker.

Benefits of using floating-point hardware versus software floating-point libraries

Code that uses floating-point hardware is more compact and faster than code that uses software libraries for floating-point arithmetic. But code that uses the floating-point hardware can only be run on processors that have the floating-point hardware. Code that uses software floating-point libraries can also run on Arm-based processors that do not have floating-point hardware, for example, the Cortex-M0 processor, and this makes the code more portable. You might also disable floating-point hardware to reduce power consumption.

Enabling and disabling the use of floating-point hardware

By default, Arm Compiler uses the available floating-point hardware that is based on the target you specify for `-mcpu` or `-march`. However, you can force Arm Compiler to disable the floating-point hardware. This forces Arm Compiler to use software floating-point libraries, if available, to perform the floating-point arithmetic in your source code.

When compiling for AArch64:

- By default, Arm Compiler uses floating-point hardware that is available on the target.
- To disable the use of floating-point arithmetic, use the `+nofp` extension on the `-mcpu` or `-march` options.

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a+nofp
```

- Software floating-point library for AArch64 is not currently available. Therefore, Arm Compiler does not support floating-point arithmetic in your source code, if you disable floating-point hardware when compiling for AArch64 targets.
- Disabling floating-point arithmetic does not disable all the floating-point hardware because the floating-point hardware is also used for Advanced SIMD arithmetic. To disable all Advanced SIMD and floating-point hardware, use the `+nofp+nosimd` extension on the `-mcpu` or `-march` options:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a+nofp+nosimd
```

See the *armclang Reference Guide* for more information on the *-march* option.

When compiling for AArch32:

- By default, Arm Compiler uses floating-point hardware that is available on the target, with the exception for Armv6-M, which does not have any floating-point hardware.
- To disable the use of floating-point hardware instructions, use the `-mfpu=none` option.

```
armclang --target=arm-arm-none-eabi -march=armv8-a -mfpu=none
```

- On AArch32 targets, using `-mfpu=none` disables the hardware for both Advanced SIMD and floating-point arithmetic. You can use `-mfpu` to selectively enable certain hardware features. For example, if you want to use the hardware for Advanced SIMD operations on an Armv7 architecture-based processor, but not for floating-point arithmetic, then use `-mfpu=neon`.

```
armclang --target=arm-arm-none-eabi -march=armv7-a -mfpu=neon
```

See the *armclang Reference Guide* for more information on the *-mfpu* option.

Floating-point linkage

Floating-point linkage refers to how the floating-point arguments are passed to and returned from function calls.

For AArch64, Arm Compiler always uses hardware floating-point registers to pass and return floating-point values. This is called hardware linkage.

For AArch32, Arm Compiler can use hardware linkage or software linkage. When using software linkage, floating-point values are passed and returned using the general purpose registers. By default, Arm Compiler uses software linkage. You can use the `-mfloat-abi` option to force hardware linkage or software linkage.

Table 2-6 Floating-point linkage for AArch32

-mfloat-abi value	Linkage	Floating-point operations
`hard`	Hardware linkage. Use floating-point registers. But if `-mfpu=none` is specified for AArch32, then use general-purpose registers.	Use hardware floating-point instructions. But if `-mfpu=none` is specified for AArch32, then use software libraries.
`soft`	Software linkage. Use general-purpose registers.	Use software libraries without floating-point hardware.
`softfp` (This is the default)	Software linkage. Use general-purpose registers.	Use hardware floating-point instructions. But if `-mfpu=none` is specified for AArch32, then use software libraries.

Code with hardware linkage can be faster than the same code with software linkage. However, code with software linkage can be more portable because it does not require the hardware floating-point registers. Hardware floating-point is not available on some architectures such as Armv6-M, or on processors where the floating-point hardware might be powered down for energy efficiency reasons.

See the *armclang Reference Guide* for more information on the *-mfloat-abi* option.

Note

All objects to be linked together must have the same type of linkage. If you link object files that have hardware linkage with object files that have software linkage, then the image might have unpredictable behavior. When linking objects, specify the `armlink` option `--fpu=`*`name`* where *`name`* specifies the correct linkage type and floating-point hardware. This enables the linker to provide diagnostic information if it detects different linkage types.

See the *armlink User Guide* for more information on how the *--fpu* option specifies the linkage type and floating-point hardware.

2.8 Compilation tools command-line option rules

You can use command-line options to control many aspects of the compilation tools' operation. There are rules that apply to each tool.

`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)'
```

Chapter 3
Writing Optimized Code

To make best use of the optimization capabilities of Arm Compiler, there are various options, pragmas, attributes, and coding techniques that you can use.

It contains the following sections:

3.1 Optimizing loops

Loops can take a significant amount of time to complete depending on the number of iterations in the loop. The overhead of checking a condition for each iteration of the loop can degrade the performance of the loop.

Loop unrolling

You can reduce the impact of this overhead by unrolling some of the iterations, which in turn reduces the number of iterations for checking the condition. Use `#pragma unroll (n)` to unroll time-critical loops in your source code. However, unrolling loops has the disadvantage of increasing the codes size. These pragmas are only effective at optimization `-O2`, `-O3`, `-Ofast`, and `-Omax`.

Table 3-1 Loop unrolling pragmas

Pragma	Description
#pragma unroll (*n*)	Unroll *n* iterations of the loop.
#pragma unroll_completely	Unroll all the iterations of the loop.

The examples below show code with loop unrolling and code without loop unrolling.

Table 3-2 Loop optimizing example

Bit counting loop without unrolling

```
int countSetBits1(unsigned int n)
{
    int bits = 0;

    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}
```

Bit counting loop with unrolling

```
int countSetBits2(unsigned int n)
{
    int bits = 0;
    #pragma unroll (4)
    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}
```

The code below shows the code that Arm Compiler generates for the above examples. Copy the examples above into `file.c` and compile using:

```
armclang --target=arm-arm-none-eabi -march=armv8-a file.c -O2 -c -S -o file.s
```

For the function with loop unrolling, `countSetBits2`, the generated code is faster but larger in size.

Table 3-3 Loop examples

Bit counting loop without unrolling

```
countSetBits1:
        mov     r1, r0
        mov     r0, #0
        cmp     r1, #0
        bxeq    lr
        mov     r2, #0
        mov     r0, #0
.LBB0_1:
        and     r3, r1, #1
        cmp     r2, r1, asr #1
        add     r0, r0, r3
        lsr     r3, r1, #1
        mov     r1, r3
        bne     .LBB0_1
        bx      lr
```

Bit counting loop with unrolling

```
countSetBits2:
        mov     r1, r0
        mov     r0, #0
        cmp     r1, #0
        bxeq    lr
        mov     r2, #0
        mov     r0, #0
LBB0_1:
        and     r3, r1, #1
        cmp     r2, r1, asr #1
        add     r0, r0, r3
        beq     .LBB0_4
@ BB#2:
        asr     r3, r1, #1
        cmp     r2, r1, asr #2
        and     r3, r3, #1
        add     r0, r0, r3
        asrne   r3, r1, #2
        andne   r3, r3, #1
        addne   r0, r0, r3
        cmpne   r2, r1, asr #3
        beq     .LBB0_4
@ BB#3:
        asr     r3, r1, #3
        cmp     r2, r1, asr #4
        and     r3, r3, #1
        add     r0, r0, r3
        asr     r3, r1, #4
        mov     r1, r3
        bne     .LBB0_1
.LBB0_4:
        bx    lr
```

Arm Compiler can unroll loops completely only if the number of iterations is known at compile time.

Loop vectorization

If your target has the Advanced SIMD unit, then Arm Compiler can use the vectorizing engine to optimize vectorizable sections of the code. At optimization level `-O1`, you can enable vectorization using `-fvectorize`. At higher optimizations, `-fvectorize` is enabled by default and you can disable it using `-fno-vectorize`. See *-fvectorize* in the *armclang Reference Guide* for more information. When using `-fvectorize` with `-O1`, vectorization might be inhibited in the absence of other optimizations which might be present at `-O2` or higher.

For example, loops that access structures can be vectorized if all parts of the structure are accessed within the same loop rather than in separate loops. The following examples show code with a loop that can be vectorized by Advanced SIMD, and a loop that cannot be vectorized easily.

Table 3-4 Example loops

Vectorizable by Advanced SIMD

```
typedef struct tBuffer {
  int a;
  int b;
  int c;
} tBuffer;
tBuffer buffer[8];

void DoubleBuffer1 (void)
{
  int i;
  for (i=0; i<8; i++)
  {
    buffer[i].a *= 2;
    buffer[i].b *= 2;
    buffer[i].c *= 2;
  }
}
```

Not vectorizable by Advanced SIMD

```
typedef struct tBuffer {
  int a;
  int b;
  int c;
} tBuffer;
tBuffer buffer[8];

void DoubleBuffer2 (void)
{
  int i;
  for (i=0; i<8; i++)
    buffer[i].a *= 2;
  for (i=0; i<8; i++)
    buffer[i].b *= 2;
  for (i=0; i<8; i++)
    buffer[i].c *= 2;
}
```

For each example above, copy the code into file.c and compile at optimization level `O2` to enable auto-vectorization:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O2 file.c -c -S -o file.s
```

The vectorized assembly code contains the Advanced SIMD instructions, for example `vld1`, `vshl`, and `vst1`. These Advanced SIMD instructions are not generated when compiling the example with the non-vectorizable loop.

Table 3-5 Assembly code from vectorizable and non-vectorizable loops

Vectorized assembly code

```
DoubleBuffer1:
.fnstart
@ BB#0:
        movw    r0, :lower16:buffer
        movt    r0, :upper16:buffer
        vld1.64 {d16, d17}, [r0:128]
        mov     r1, r0
        vshl.i32        q8, q8, #1
        vst1.32 {d16, d17}, [r1:128]!
        vld1.64 {d16, d17}, [r1:128]
        vshl.i32        q8, q8, #1
        vst1.64 {d16, d17}, [r1:128]
        add     r1, r0, #32
        vld1.64 {d16, d17}, [r1:128]
        vshl.i32        q8, q8, #1
        vst1.64 {d16, d17}, [r1:128]
        add     r1, r0, #48
        vld1.64 {d16, d17}, [r1:128]
        vshl.i32        q8, q8, #1
        vst1.64 {d16, d17}, [r1:128]
        add     r1, r0, #64
        add     r0, r0, #80
        vld1.64 {d16, d17}, [r1:128]
        vshl.i32        q8, q8, #1
        vst1.64 {d16, d17}, [r1:128]
        vld1.64 {d16, d17}, [r0:128]
        vshl.i32        q8, q8, #1
        vst1.64 {d16, d17}, [r0:128]
        bxlr
```

Non-vectorized assembly code

```
DoubleBuffer2:
    .fnstart
@ BB#0:
           movw    r0, :lower16:buffer
        movt    r0, :upper16:buffer
        ldr     r1, [r0]
        lsl     r1, r1, #1
        str     r1, [r0]
        ldr     r1, [r0, #12]
        lsl     r1, r1, #1
        str     r1, [r0, #12]
        ldr     r1, [r0, #24]
        lsl     r1, r1, #1
        str     r1, [r0, #24]
        ldr     r1, [r0, #36]
        lsl     r1, r1, #1
        str     r1, [r0, #36]
        ldr     r1, [r0, #48]
        lsl     r1, r1, #1
        str     r1, [r0, #48]
        ldr     r1, [r0, #60]
        lsl     r1, r1, #1
        str     r1, [r0, #60]
        ldr     r1, [r0, #72]
        lsl     r1, r1, #1
        str     r1, [r0, #72]
        ldr     r1, [r0, #84]
        lsl     r1, r1, #1
        str     r1, [r0, #84]
        ldr     r1, [r0, #4]
        lsl     r1, r1, #1
        str     r1, [r0, #4]
        ldr     r1, [r0, #16]
        lsl     r1, r1, #1
        ...
        bx    lr
```

Note

Advanced SIMD (Single Instruction Multiple Data), also known as Arm NEON™ technology, is a powerful vectorizing unit on Armv7-A and later Application profile architectures. It enables you to write highly optimized code. You can use intrinsics to directly use the Advanced SIMD capabilities from C or C++ code. The intrinsics and their data types are defined in `arm_neon.h`. For more information on Advanced SIMD, see the *Arm® C Language Extensions*, *Cortex®-A Series Programmer's Guide*, and *Arm® NEON™ Programmer's Guide*.

Using `-fno-vectorize` does not necessarily prevent the compiler from emitting Advanced SIMD instructions. The compiler or linker might still introduce Advanced SIMD instructions, such as when linking libraries that contain these instructions.

To prevent the compiler from emitting Advanced SIMD instructions for AArch64 targets, specify `+nosimd` using `-march` or `-mcpu`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a+nosimd -O2 file.c -c -S -o file.s
```

To prevent the compiler from emitting Advanced SIMD instructions for AArch32 targets, set the option `-mfpu` to the correct value that does not include Advanced SIMD, for example `fp-armv8`.

```
armclang --target=aarch32-arm-none-eabi -march=armv8-a -mfpu=fp-armv8 -O2 file.c -c -S -o
file.s
```

Related information

armclang -O option.

pragma unroll.

armclang -fvectorize option.

3.2 Inlining functions

Arm Compiler automatically inlines functions if it decides that inlining the function gives better performance. This inlining does not significantly increase the code size. However, you can use compiler hints and options to influence or control whether a function is inlined or not.

Table 3-6 Function inlining

Inlining options, keywords, or attributes	Description
__inline__	Specify this keyword on a function definition or declaration as a hint to the compiler to favor inlining of the function. However, for each function call, the compiler still decides whether to inline the function. This is equivalent to `__inline`.
__attribute__((always_inline))	Specify this function attribute on a function definition or declaration to tell the compiler to always inline this function, with certain exceptions such as for recursive functions. This overrides the `-fno-inline-functions` option.
__attribute__((noinline))	Specify this function attribute on a function definition or declaration to tell the compiler to not inline the function. This is equivalent to `__declspec(noinline)`.
-fno-inline-functions	This is a compiler command-line option. Specify this option to the compiler to disable inlining. This option overrides the `__inline__` hint.

Note

- Arm Compiler only inlines functions within the same compilation unit, unless you use Link Time Optimization. For more information, see *Optimizing across modules with link time optimization* in the *Software Development Guide*.
- C++ and C99 provide the `inline` language keyword. The effect of this `inline` language keyword is identical to the effect of using the `__inline__` compiler keyword. However, the effect in C99 mode is different from the effect in C++ or other C that does not adhere to the C99 standard. For more information, see *Inline functions* in the *armclang Reference Guide*.
- Function inlining normally happens at higher optimization levels, such as `-O2`, except when you specify `__attribute__((always_inline))`.

Examples of function inlining

This example shows the effect of __attribute__((always_inline)) and `-fno-inline-functions` in C99 mode, which is the default behavior for C files. Copy the following code to `file.c`.

```
int bar(int a)
{
    a=a*(a+1);
    return a;
}

__attribute__((always_inline)) static int row(int a)
{
    a=a*(a+1);
    return a;
}

int foo (int i)
{
    i=bar(i);
    i=i-2;
    i=bar(i);
    i++;
    i=row(i);
    i++;
    return i;
}
```

In the example code, functions `bar` and `row` are identical but function `row` is always inlined. Use the following compiler commands to compile for `-O2` with `-fno-inline-functions` and without `-fno-inline-functions`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c -S file.c -O2 -o file_no_inline.s -fno-inline-functions
```

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c -S file.c -O2 -o file_with_inline.s
```

The generated code shows inlining:

Table 3-7 Effect of -fno-inline-functions

Compiling with -fno-inline-functions

```
foo:                                    @ @foo
        .fnstart
@ BB#0:
        .save   {r11, lr}
        push    {r11, lr}
        bl     bar
        sub     r0, r0, #2
        bl     bar
        add     r1, r0, #1
        add     r0, r0, #2
        mul     r0, r0, r1
        add     r0, r0, #1
        pop     {r11, pc}
.Lfunc_end0:
       .size   foo, .Lfunc_end0-foo
        .cantunwind
        .fnend
```

Compiling without -fno-inline-functions

```
foo:                                    @ @foo
        .fnstart
@ BB#0:
        add     r1, r0, #1
        mul     r0, r1, r0
        sub     r1, r0, #2
        sub     r0, r0, #1
        mul     r0, r0, r1
        add     r1, r0, #1
        add     r0, r0, #2
        mul     r0, r0, r1
        add     r0, r0, #1
        bx    lr
.Lfunc_end0:
       .size   foo, .Lfunc_end0-foo
        .cantunwind
        .fnend
```

When compiling with `-fno-inline-functions`, the compiler does not inline the function `bar`. When compiling without `-fno-inline-functions`, the compiler inlines the function `bar`. However, the compiler always inlines the function `row` even though it is identical to function `bar`.

Related information

armclang -fno-inline-functions option.
__inline keyword.
__attribute__((always_inline)) function attribute.
__attribute__((no_inline)) function attribute.

3.3 Examining stack usage

Processors for embedded applications have limited memory and therefore the amount of space available on the stack is also limited. You can use Arm Compiler to determine how much stack space is used by the functions in your application code.

The amount of stack that a function uses depends on factors such as the number and type of arguments to the function, local variables in the function, and the optimizations that the compiler performs. For more information on what the stack is used for, see *Stack use in C and C++* in the *Software Development Guide*.

It is good practice to examine the amount of stack used by the functions in your application. You can then consider rewriting your code to reduce stack usage.

To examine the stack usage in your application, use the linker option `--info=stack`. The following example code shows functions with different numbers of arguments:

```
__attribute__((noinline)) int fact(int n)
{
  int f = 1;
  while (n>0)
      f *= n--;
  return f;
}

int foo (int n)
{
  return fact(n);
}

int foo_mor (int a, int b, int c, int d)
{
 return fact(a);
}

int main (void)
{
  return foo(10) + foo_mor(10,11,12,13);
}
```

Copy the code example to `file.c` and compile it using the following command:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c -g file.c -o file.o
```

Compiling with the `-g` option generates the DWARF frame information that `armlink` requires for estimating the stack use. Run `armlink` on the object file using `--info=stack`:

```
armlink file.o --info=stack
```

For the example code, `armlink` shows the amount of stack used by the various functions. Function `foo_mor` has more arguments than function `foo`, and therefore uses more stack.

```
Stack Usage for fact 0xc bytes.
Stack Usage for foo 0x8 bytes.
Stack Usage for foo_mor 0x10 bytes.
Stack Usage for main 0x8 bytes.
```

You can also examine stack usage using the linker option `--callgraph`:

```
armlink file.o --callgraph -o FileImage.axf
```

This outputs a file called `FileImage.htm` which contains the stack usage information for the various functions in the application.

```
fact (ARM, 84 bytes, Stack size 12 bytes, file.o(.text))

[Stack]

Max Depth = 12
Call Chain = fact

[Called By]
```

```
>>   foo_mor
>>   foo
foo (ARM, 36 bytes, Stack size 8 bytes, file.o(.text))

[Stack]

Max Depth = 20
Call Chain = foo >> fact

[Calls]
>>   fact

[Called By]
>>   main
foo_mor (ARM, 76 bytes, Stack size 16 bytes, file.o(.text))

[Stack]

Max Depth = 28
Call Chain = foo_mor >> fact

[Calls]
>>   fact

[Called By]
>>   main
main (ARM, 76 bytes, Stack size 8 bytes, file.o(.text))

[Stack]

Max Depth = 36
Call Chain = main >> foo_mor >> fact

[Calls]
>>   foo_mor
>>   foo

[Called By]
>>   __rt_entry_main (via BLX)
```

See *--info* and *--callgraph* in the *armlink User Guide* for more information on these options.

3.4 Packing data structures

You can reduce the amount of memory that your application requires by packing data into structures. This is especially important if you need to store and access large arrays of data in embedded systems.

If individual data members in a structure are not packed, the compiler can add padding within the structure for faster access to individual members, based on the natural alignment of each member. Arm Compiler 6 provides a pragma and attribute to pack the members in a structure or union without any padding.

Table 3-8 Packing members in a structure or union

Pragma or attribute	Description
`#pragma pack (n)`	For each member, if *n* bytes is less than the natural alignment of the member, then set the alignment to *n* bytes, otherwise the alignment is the natural alignment of the member. For more information see *#pragma pack (n)* and *__alignof__*.
`__attribute__((packed))`	This is equivalent to `#pragma pack (1)`. However, the attribute can also be used on individual members in a structure or union.

Packing the entire structure

To pack the entire structure or union, use `__attribute__((packed))` or `#pragma pack(n)` to the declaration of the structure as shown in the code examples. The attribute and pragma apply to all the members of the structure or union. If the member is a structure, then the structure has an alignment of 1-byte, but the members of that structure continue to have their natural alignment.

When using `#pragma pack(n)`, the alignment of the structure is the alignment of the largest member after applying `#pragma pack(n)` to the structure.

Each example declares two objects `c` and `d`. Copy each example into `file.c` and compile:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c file.c -o file.o
```

For each example use linker option `--info=sizes` to examine the memory used in `file.o`.

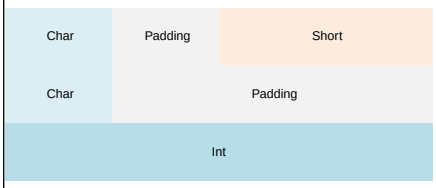
```
armlink file.o --info=sizes
```

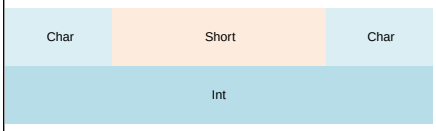
The linker output shows the total memory used by the two objects `c` and `d`. For example:

```
Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name
  36          0          0          0         24          0   str.o
--------------------------------------------------------------------------------
  36          0         16          0         24          0   Object Totals
```

Table 3-9 Packing structures

Code	Packing	Size of structure

`struct stc`
`{`
`  char one;`
`  short two;`
`  char three;`
`  int four;`
`} c,d;`
 	
`int main (void)`
`{`
`  c.one=1;`
`  return 0;`
`}`
	 **Figure 3-1 Structure without packing attribute or pragma**	12. The alignment of the structure is the natural alignment of the largest member. In this example, the largest member is an `int`.

`struct __attribute__((packed))`
`stc`
`{`
`  char one;`
`  short two;`
`  char three;`
`  int four;`
`} c,d;`
 	
`int main (void)`
`{`
`  c.one=1;`
`  return 0;`
`}`
	 **Figure 3-2 Structure with attribute packed**	8. The alignment of the structure is 1 byte.

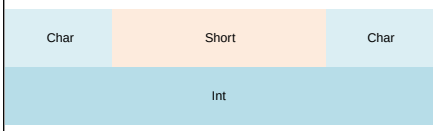
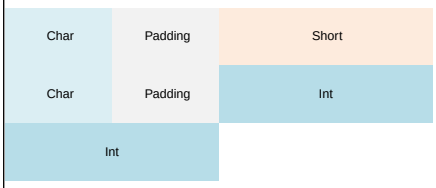
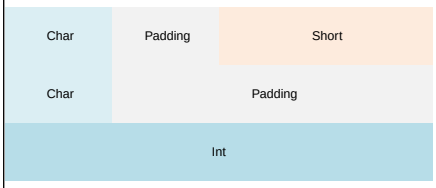
`#pragma pack (1)`
`struct stc`
`{`
`  char one;`
`  short two;`
`  char three;`
`  int four;`
`} c,d;`
 	
`int main (void)`
`{`
`  c.one=1;`
`  return 0;`
`}`
	 **Figure 3-3 Structure with pragma pack with 1 byte alignment**	8. The alignment of the structure is 1 byte.

Table 3-9 Packing structures (continued)

Code	Packing	Size of structure

`#pragma pack (2)`
`struct stc`
`{`
`  char one;`
`  short two;`
`  char three;`
`  int four;`
`} c,d;`
 	
`int main (void)`
`{`
`  c.one=1;`
`  return 0;`
`}`
	 **Figure 3-4 Structure with pragma pack with 2 byte alignment**	10. The alignment of the structure is 2 bytes.

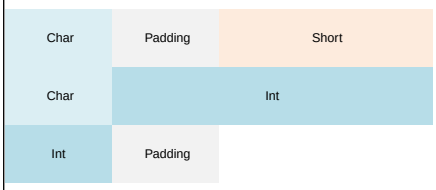
`#pragma pack (4)`
`struct stc`
`{`
`  char one;`
`  short two;`
`  char three;`
`  int four;`
`} c,d;`
 	
`int main (void)`
`{`
`  c.one=1;`
`  return 0;`
`}`
	 **Figure 3-5 Structure with pragma pack with 4 byte alignment**	12. The alignment of the structure is 4 bytes.

Packing individual members in a structure

To pack individual members of a structure, use `__attribute__((packed))` on the member. This aligns the member to a byte boundary and therefore reduces the amount of memory required by the structure as a whole. It does not affect the alignment of the other members. Therefore the alignment of the whole structure is equal to the alignment of the largest member without the `__attribute__((packed))`.

Table 3-10 Packing individual members

Code	Packing	Size

`struct stc`
`{`
`  char one;`
`  short two;`
`  char three;`
`  int  __attribute__((packed))`
`four;`
`} c,d;`
 	
`int main (void)`
`{`
`  c.one=1;`
`  return 0;`
`}`
	 **Figure 3-6 Structure with attribute packed on individual member**	10. The alignment of the structure is 2 bytes because the largest member without `__attribute__((packed))` is `short`.

Accessing packed members from a structure

If a member of a structure or union is packed and therefore does not have its natural alignment, then to access this member, you must use the structure or union that contains this member. You must not take the address of such a packed member to use as a pointer, because the pointer might be unaligned.

Dereferencing such a pointer can be unsafe even when unaligned accesses are supported by the target, because certain instructions always require word-aligned addresses.

Note

If you take the address of a packed member, in most cases, the compiler generates a warning.

```
struct __attribute__((packed)) foobar
{
  char x;
  short y;
};

short get_y(struct foobar *s)
{
    // Correct usage: the compiler will not use unaligned accesses
    // unless they are allowed.
    return s->y;
}

short get2_y(struct foobar *s)
{
    short *p = &s->y; // Incorrect usage: 'p' might be an unaligned pointer.
    return *p;  // This might cause an unaligned access.
}
```

Related information

pragma pack.
__attribute__((packed)) type attribute.
__attribute__((packed)) variable attribute.

Chapter 4
Using Assembly and Intrinsics in C or C++ Code

All code for a single application can be written in the same source language. This is usually a high-level language such as C or C++ that is compiled to instructions for Arm architectures. However, in some situations you might need lower-level control than what C and C++ provide.

For example:

- To access features which are not available from C or C++, such as interfacing directly with device hardware.
- To generate highly optimized code by manually writing sections using intrinsics or inline assembly.

There are a number of different ways to have low-level control over the generated code:

- Intrinsics are functions provided by the compiler. An intrinsic function has the appearance of a function call in C or C++, but is replaced during compilation by a specific sequence of low-level instructions.
- Inline assembly lets you write assembly instructions directly in your C/C++ code, without the overhead of a function call.
- Calling assembly functions from C/C++ lets you write standalone assembly code in a separate source file. This code is assembled separately to the C/C++ code, and then integrated at link time.

It contains the following sections:

4.1 Using intrinsics

Compiler intrinsics are special functions whose implementations are known to the compiler. They enable you to easily incorporate domain-specific operations in C and C++ source code without resorting to complex implementations in assembly language.

The C and C++ languages are suited to a wide variety of tasks but they do not provide built-in support for specific areas of application, for example *Digital Signal Processing* (DSP).

Within a given application domain, there is usually a range of domain-specific operations that have to be performed frequently. However, if specific hardware support is available, then these operations can often be implemented more efficiently using the hardware support than in C or C++. A typical example is the saturated add of two 32-bit signed two's complement integers, commonly used in DSP programming. The following example shows a C implementation of a saturated add operation:

```
#include <limits.h>
int L_add(const int a, const int b)
{
    int c;
    c = a + b;
    if (((a ^ b) & INT_MIN) == 0)
    {
        if ((c ^ a) & INT_MIN)
        {
            c = (a < 0) ? INT_MIN : INT_MAX;
        }
    }
    return c;
}
```

Using compiler intrinsics, you can achieve more complete coverage of target architecture instructions than you would from the instruction selection of the compiler.

An intrinsic function has the appearance of a function call in C or C++, but is replaced during compilation by a specific sequence of low-level instructions. The following example shows how to access the `__qadd` saturated add intrinsic:

```
#include <arm_acle.h>    /* Include ACLE intrinsics */

int foo(int a, int b)
{
  return __qadd(a, b);  /* Saturated add of a and b */
}
```

Using compiler intrinsics offers a number of performance benefits:

- The low-level instructions substituted for an intrinsic are either as efficient or more efficient than corresponding implementations in C or C++. This results in both reduced instruction and cycle counts. To implement the intrinsic, the compiler automatically generates the best sequence of instructions for the specified target architecture. For example, the `__qadd` intrinsic maps directly to the A32 assembly language instruction `qadd`:

  ```
  QADD r0, r0, r1    /* Assuming r0 = a, r1 = b on entry */
  ```

- More information is given to the compiler than the underlying C and C++ language is able to convey. This enables the compiler to perform optimizations and to generate instruction sequences that it cannot otherwise perform.

These performance benefits can be significant for real-time processing applications. However, care is required because the use of intrinsics can decrease code portability.

Related information

Compiler-specific intrinsics.

ACLE support.

4.2 Writing inline assembly code

The compiler provides an inline assembler that enables you to write assembly code in your C or C++ source code, for example to access features of the target processor that are not available from C or C++.

The `__asm` keyword can incorporate inline assembly code into a function using the GNU inline assembly syntax. For example:

```
#include <stdio.h>

int add(int i, int j)
{
  int res = 0;
  __asm ("ADD %[result], %[input_i], %[input_j]"
    : [result] "=r" (res)
    : [input_i] "r" (i), [input_j] "r" (j)
  );
  return res;
}

int main(void)
{
  int a = 1;
  int b = 2;
  int c = 0;

  c = add(a,b);

  printf("Result of %d + %d = %d\n", a, b, c);
}
```

Note

The inline assembler does not support legacy assembly code written in `armasm` assembler syntax. See the *Migration and Compatibility Guide* for more information about migrating `armasm` syntax assembly code to GNU syntax.

The general form of an `__asm` inline assembly statement is:

```
__asm [volatile] (code); /* Basic inline assembly syntax */
```

```
/* Extended inline assembly syntax */
__asm [volatile] (code_template
       : output_operand_list
      [: input_operand_list
      [: clobbered_register_list]]
  );
```

code is the assembly instruction, for example `"ADD R0, R1, R2"`. *code_template* is a template for an assembly instruction, for example `"ADD %[result], %[input_i], %[input_j]"`.

If you specify a `code_template` rather than *code* then you must specify the *output_operand_list* before specifying the optional *input_operand_list* and *clobbered_register_list*.

output_operand_list is a list of output operands, separated by commas. Each operand consists of a symbolic name in square brackets, a constraint string, and a C expression in parentheses. In this example, there is a single output operand: `[result] "=r" (res)`. The list can be empty. For example:

```
__asm ("ADD R0, %[input_i], %[input_j]"
    :  /* This is an empty output operand list */
    : [input_i] "r" (i), [input_j] "r" (j)
  );
```

input_operand_list is an optional list of input operands, separated by commas. Input operands use the same syntax as output operands. In this example, there are two input operands: `[input_i] "r" (i)`, `[input_j] "r" (j)`. The list can be empty.

clobbered_register_list is an optional list of clobbered registers whose contents are not preserved. The list can be empty. In addition to registers, the list can also contain special arguments:

"cc"
The instruction affects the condition code flags.

"memory"
The instruction accesses unknown memory addresses.

The registers in *clobbered_register_list* must use lowercase letters rather than uppercase letters. An example instruction with a *clobbered_register_list* is:

```
__asm ("ADD R0, %[input_i], %[input_j]"
    :  /* This is an empty output operand list */
    : [input_i] "r" (i), [input_j] "r" (j)
    : "r5","r6","cc","memory" /*Use "r5" instead of "R5" */
  );
```

Use the `volatile` qualifier for assembler instructions that have processor side-effects, which the compiler might be unaware of. The `volatile` qualifier disables certain compiler optimizations. The `volatile` qualifier is optional.

Defining symbols and labels

You can use inline assembly to define symbols. For example:

```
__asm (".global __use_no_semihosting\n\t");
```

To define labels, use `:` after the label name. For example:

```
__asm ("my_label:\n\t");
```

Multiple instructions

You can write multiple instructions within the same `__asm` statement. This example shows an interrupt handler written in one `__asm` statement for an Armv8-M mainline architecture.

```
void HardFault_Handler(void)
{
  asm (
    "TST LR, #0x40\n\t"
    "BEQ from_nonsecure\n\t"
  "from_secure:\n\t"
    "TST LR, #0x04\n\t"
    "ITE EQ\n\t"
    "MRSEQ R0, MSP\n\t"
    "MRSNE R0, PSP\n\t"
    "B hard_fault_handler_c\n\t"
  "from_nonsecure:\n\t"
    "MRS R0, CONTROL_NS\n\t"
    "TST R0, #2\n\t"
    "ITE EQ\n\t"
    "MRSEQ R0, MSP_NS\n\t"
    "MRSNE R0, PSP_NS\n\t"
    "B hard_fault_handler_c\n\t"
  );
}
```

Copy the above handler code to `file.c` and then you can compile it using:

```
armclang --target=arm-arm-none-eabi -march=armv8-m.main -c -S file.c -o file.s
```

Embedded assembly

You can write embedded assembly using `__attribute__((naked))`. For more information, see *__attribute__((naked))* in the *armclang Reference Guide*.

4.3 Calling assembly functions from C and C++

Often, all the code for a single application is written in the same source language. This is usually a high-level language such as C or C++. That code is then compiled to Arm assembly code.

However, in some situations you might want to make function calls from C/C++ code to assembly code. For example:

- If you want to make use of existing assembly code, but the rest of your project is in C or C++.
- If you want to manually write critical functions directly in assembly code that can produce better optimized code than compiling C or C++ code.
- If you want to interface directly with device hardware and if this is easier in low-level assembly code than high-level C or C++.

Note

For code portability, it is better to use intrinsics or inline assembly rather than writing and calling assembly functions.

To call an assembly function from C or C++:

1. In the assembly source, declare the code as a global function using `.globl` and `.type`:

```
     .globl   myadd
     .p2align 2
     .type    myadd,%function

myadd:                      // Function "myadd" entry point.
     .fnstart
     add      r0, r0, r1    // Function arguments are in R0 and R1. Add together and put
the result in R0.
     bx       lr            // Return by branching to the address in the link register.
     .fnend
```

   Note

   `armclang` requires that you explicitly specify the types of exported symbols using the `.type` directive. If the `.type` directive is not specified in the above example, the linker outputs warnings of the form:

```
Warning: L6437W: Relocation #RELA:1 in test.o(.text) with respect to myadd...

Warning: L6318W: test.o(.text) contains branch to a non-code symbol myadd.
```

2. In C code, declare the external function using `extern`:

```
#include <stdio.h>

extern int myadd(int a, int b);

int main()
{
    int a = 4;
    int b = 5;
    printf("Adding %d and %d results in %d\n", a, b, myadd(a, b));
    return (0);
}
```

   In C++ code, use `extern "C"`:

```
extern "C" int myadd(int a, int b);
```

3. Ensure that your assembly code complies with the *Procedure Call Standard for the Arm® Architecture (AAPCS)*.
   The AAPCS describes a contract between caller functions and callee functions. For example, for integer or pointer types, it specifies that:
   - Registers R0-R3 pass argument values to the callee function, with subsequent arguments passed on the stack.
   - Register R0 passes the result value back to the caller function.

- Caller functions must preserve R0-R3 and R12, because these registers are allowed to be corrupted by the callee function.
- Callee functions must preserve R4-R11 and LR, because these registers are not allowed to be corrupted by the callee function.

For more information, see the *Procedure Call Standard for the Arm® Architecture (AAPCS)*.

4. Compile both source files:

```
armclang --target=arm-arm-none-eabi -march=armv8-a main.c myadd.s
```

Related information

Procedure Call Standard for the Arm Architecture.
Procedure Call Standard for the Arm 64-bit Architecture.

Chapter 5
Mapping Code and Data to the Target

There are various options in Arm Compiler to control how code, data and other sections of the image are mapped to specific locations on the target.

It contains the following sections:

5.1 What the linker does to create an image

The linker takes object files that a compiler or assembler produces and combines them into an executable image. The linker also uses a memory description to assign the input code and data from the object files to the required addresses in the image.

You can specify object files directly on the command line or specify a user library containing object files. The linker:

- Resolves symbolic references between the input object files.
- Extracts object modules from libraries to resolve otherwise unresolved symbolic references.
- Removes unused sections.
- Eliminates duplicate common groups and common code, data, and debug sections.
- Sorts input sections according to their attributes and names, and merges sections with similar attributes and names into contiguous chunks.
- Organizes object fragments into memory regions according to the grouping and placement information that is provided in a memory description.
- Assigns addresses to relocatable values.
- Generates either a partial object if requested, for input to another link step, or an executable image.

The linker has a built-in memory description that it uses by default. However, you can override this default memory description with command-line options or with a scatter file. The method that you use depends how much you want to control the placement of the various output sections in the image:

- Allow the linker to automatically place the output sections using the default memory map for the specified linking model. `armlink` uses default locations for the RO, RW, *execute-only* (XO), and ZI output sections.
- Use the memory map related command-line options to specify the locations of the RO, RW, XO, and ZI output sections.
- Use a scatter file if you want to have the most control over where the linker places various parts of your image. For example, you can place individual functions at specific addresses or certain data structures at peripheral addresses.

Note

XO sections are supported only for images that are targeted at Armv7-M or Armv8-M architectures.

This section contains the following subsection:

- *5.1.1 What you can control with a scatter file* on page 5-62.

5.1.1 What you can control with a scatter file

A scatter file gives you the ability to control where the linker places different parts of your image for your particular target.

You can control:

- The location and size of various memory regions that are mapped to ROM, RAM, and FLASH.
- The location of individual functions and variables, and code from the Arm standard C and C++ libraries.
- The placement of sections that contain individual functions or variables, or code from the Arm standard C and C++ libraries.
- The priority ordering of memory areas for placing unassigned sections, to ensure that they get filled in a particular order.
- The location and size of empty regions of memory, such as memory to use for stack and heap.

If the location of some code or data lies outside all the regions that are specified in your scatter file, the linker attempts to create a load and execution region to contain that code or data.

Note

Multiple code and data sections cannot occupy the same area of memory, unless you place them in separate overlay regions.

5.2 Placing data items for target peripherals with a scatter file

To access the peripherals on your target, you must locate the data items that access them at the addresses of those peripherals.

To make sure that the data items are placed at the correct address for the peripherals, use the `__attribute__((section(".ARM.__at_address")))` variable attribute together with a scatter file.

Procedure

1. Create `peripheral.c` to place the `my_peripheral` variable at address `0x10000000`.

```
#include "stdio.h"

int my_peripheral __attribute__((section(".ARM.__at_0x10000000"))) = 0;

int main(void)
{
    printf("%d\n",my_peripheral);
    return 0;
}
```

2. Create the scatter file `scatter.scat`.

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        *(+RO +RW)     ; all RO sections (must include section with
                       ; initial entry point)
    }
    ; rest of scatter-loading description

    ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
}

LR_2 0x01000000
{
    ER_ZI +0 UNINIT
    {
        *(.bss)
    }
}

LR_3 0x10000000
{
    ER_PERIPHERAL 0x10000000 UNINIT
    {
        *(.ARM.__at_0x10000000)
    }
}
```

3. Build the image.

```
armclang --target=arm-arm-eabi-none -mcpu=cortex-a9 peripheral.c -g -c -o peripheral.o
armlink --cpu=cortex-a9 --scatter=scatter.scat --map --symbols peripheral.o --
output=peripheral.axf > map.txt
```

The memory map for load region `LR_3` is:

```
  Load Region LR_3 (Base: 0x10000000, Size: 0x00000004, Max: 0xffffffff, ABSOLUTE)

    Execution Region ER_PERIPHERAL (Base: 0x10000000, Size: 0x00000004, Max: 0xffffffff,
ABSOLUTE, UNINIT)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x10000000   0x00000004   Data   RW            5    .ARM.__at_0x10000000  peripheral.o
```

5.3 Placing the stack and heap with a scatter file

The Arm C library provides multiple implementations of the function `__user_setup_stackheap()`, and can select the correct one for you automatically from information that is given in a scatter file.

Note

- If you re-implement `__user_setup_stackheap()` then your version does not get invoked when stack and heap are defined in a scatter file.
- You might have to update your startup code to use the correct initial stack pointer. Some processors, such as the Cortex-M3 processor, require that you place the initial stack pointer in the vector table. See *Stack and heap configuration* in *AN179 - Cortex®-M3 Embedded Software Development* for more details.

Procedure

1. Define two special execution regions in your scatter file that is named `ARM_LIB_HEAP` and `ARM_LIB_STACK`.
2. Assign the `EMPTY` attribute to both regions.

   Because the stack and heap are in separate regions, the library selects the non-default implementation of `__user_setup_stackheap()` that uses the value of the symbols:

   - `Image$$ARM_LIB_STACK$$ZI$$Base`.
   - `Image$$ARM_LIB_STACK$$ZI$$Limit`.
   - `Image$$ARM_LIB_HEAP$$ZI$$Base`.
   - `Image$$ARM_LIB_HEAP$$ZI$$Limit`.

   You can specify only one `ARM_LIB_STACK` or `ARM_LIB_HEAP` region, and you must allocate a size.

   ```
   LOAD_FLASH …
   {
       …
       ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
       { }
       ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
       { }
       …
   }
   ```

3. Alternatively, define a single execution region that is named `ARM_LIB_STACKHEAP` to use a combined stack and heap region. Assign the `EMPTY` attribute to the region.

   Because the stack and heap are in the same region, `__user_setup_stackheap()` uses the value of the symbols `Image$$ARM_LIB_STACKHEAP$$ZI$$Base` and `Image$$ARM_LIB_STACKHEAP$$ZI$$Limit`.

5.4 Root region

A root region is a region with the same load and execution address. The initial entry point of an image must be in a root region.

If the initial entry point is not in a root region, the link fails and the linker gives an error message.

Example

Root region with the same load and execution address.

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        * (+RO)        ; all RO sections (must include section with
        ; initial entry point)
    }
    …                ; rest of scatter-loading description
}
```

This section contains the following subsections:

- *5.4.1 Effect of the ABSOLUTE attribute on a root region* on page 5-66.
- *5.4.2 Effect of the FIXED attribute on a root region* on page 5-67.

5.4.1 Effect of the ABSOLUTE attribute on a root region

You can use the `ABSOLUTE` attribute to specify a root region. This attribute is the default for an execution region.

To specify a root region, use `ABSOLUTE` as the attribute for the execution region. You can either specify the attribute explicitly or permit it to default, and use the same address for the first execution region and the enclosing load region.

To make the execution region address the same as the load region address, either:

- Specify the same numeric value for both the base address for the execution region and the base address for the load region.
- Specify a `+0` offset for the first execution region in the load region.

  If you specify an offset of zero (`+0`) for all subsequent execution regions in the load region, then all execution regions not following an execution region containing ZI are also root regions.

Example

The following example shows an implicitly defined root region:

```
LR_1 0x040000                  ; load region starts at 0x40000
{                              ; start of execution region descriptions
    ER_RO 0x040000 ABSOLUTE    ; load address = execution address
    {
        * (+RO)                ; all RO sections (must include the section
                               ; containing the initial entry point)
    }
    …                        ; rest of scatter-loading description
}
```

5.4.2 Effect of the FIXED attribute on a root region

You can use the `FIXED` attribute for an execution region in a scatter file to create root regions that load and execute at fixed addresses.

Use the `FIXED` execution region attribute to ensure that the load address and execution address of a specific region are the same.

You can use the `FIXED` attribute to place any execution region at a specific address in ROM.

For example, the following memory map shows fixed execution regions:

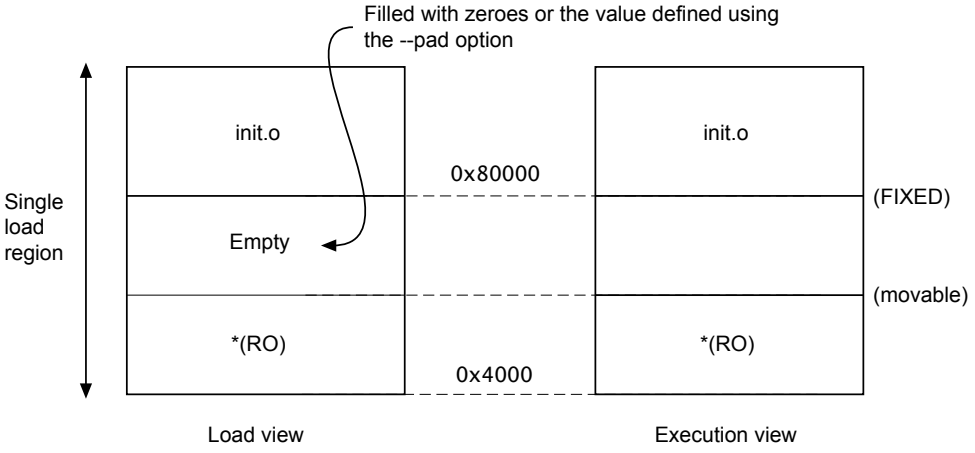


Figure 5-1 Memory map for fixed execution regions

The following example shows the corresponding scatter-loading description:

```
LR_1 0x040000              ; load region starts at 0x40000
{                          ; start of execution region descriptions
    ER_RO 0x040000         ; load address = execution address
    {
        * (+RO)            ; RO sections other than those in init.o
    }
    ER_INIT 0x080000 FIXED ; load address and execution address of this
                           ; execution region are fixed at 0x80000
    {
        init.o(+RO)        ; all RO sections from init.o
    }
    …                    ; rest of scatter-loading description
}
```

You can use this to place a function or a block of data, such as a constant table or a checksum, at a fixed address in ROM so that it can be accessed easily through pointers.

If you specify, for example, that some initialization code is to be placed at start of ROM and a checksum at the end of ROM, some of the memory contents might be unused. Use the `*` or `.ANY` module selector to flood fill the region between the end of the initialization block and the start of the data block.

To make your code easier to maintain and debug, it is suggested that you use the minimum amount of placement specifications in scatter files and leave the detailed placement of functions and data to the linker.

Note

There are some situations where using `FIXED` and a single load region are not appropriate. Other techniques for specifying fixed locations are:

- If your loader can handle multiple load regions, place the RO code or data in its own load region.
- If you do not require the function or data to be at a fixed location in ROM, use `ABSOLUTE` instead of `FIXED`. The loader then copies the data from the load region to the specified address in RAM. `ABSOLUTE` is the default attribute.
- To place a data structure at the location of memory-mapped I/O, use two load regions and specify `UNINIT`. `UNINIT` ensures that the memory locations are not initialized to zero.

Example showing the misuse of the FIXED attribute

The following example shows common cases where the `FIXED` execution region attribute is misused:

```
LR1 0x8000
{
    ER_LOW +0 0x1000
    {
        *(+RO)
    }
; At this point the next available Load and Execution address is 0x8000 + size of
; contents of ER_LOW. The maximum size is limited to 0x1000 so the next available Load
; and Execution address is at most 0x9000
    ER_HIGH 0xF0000000 FIXED
    {
        *(+RW,+ZI)
    }
; The required execution address and load address is 0xF0000000. The linker inserts
; 0xF0000000 - (0x8000 + size of(ER_LOW)) bytes of padding so that load address matches
; execution address
}
; The other common misuse of FIXED is to give a lower execution address than the next
; available load address.
LR_HIGH 0x100000000
{
    ER_LOW 0x1000 FIXED
    {
        *(+RO)
    }
; The next available load address in LR_HIGH is 0x10000000. The required Execution
; address is 0x1000. Because the next available load address in LR_HIGH must increase
; monotonically the linker cannot give ER_LOW a Load Address lower than 0x10000000
}
```

5.5 Placing functions and data in a named section

You can place functions and data by separating them into their own objects without having to use toolchain-specific pragmas or attributes. Alternatively, you can specify a name of a section using the function or variable attribute, `__attribute__((section("name")))`.

You can use `__attribute__((section("name")))` to place a function or variable in a separate ELF section, where *name* is a name of your choice. You can then use a scatter file to place the named sections at specific locations.

You can place ZI data in a named section with `__attribute__((section(".bss.name")))`.

Use the following procedure to modify your source code to place functions and data in a specific section using a scatter file.

Procedure

1. Create a C source file `file.c` to specify a section name `foo` for a variable and a section name `.bss.mybss` for a zero-initialized variable `z`, for example:

```
#include "stdio.h"

int variable __attribute__((section("foo"))) = 10;
__attribute__((section(".bss.mybss"))) int z;

int main(void)
{
    int x = 4;
    int y = 7;
    z = x + y;
    printf("%d\n",variable);
    printf("%d\n",z);
    return 0;
}
```

2. Create a scatter file to place the named section, `scatter.scat`, for example:

```
LR_1 0x0
{
    ER_RO 0x0 0x4000
    {
        *(+RO)
    }
    ER_RW 0x4000 0x2000
    {
        *(+RW)
    }
    ER_ZI 0x6000 0x2000
    {
        *(+ZI)
    }
    ER_MYBSS 0x8000 0x2000
    {
        *(.bss.mybss)
    }

    ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
}

FLASH 0x24000000 0x4000000
{
    ; rest of code

    ADDER 0x08000000
    {
        file.o (foo)                 ; select section foo from file.o
    }

}
```

The ARM_LIB_STACK and ARM_LIB_HEAP regions are required because the program is being linked with the semihosting libraries.

Note

If you omit file.o (foo) from the scatter file, the linker places the section in the region of the same type. That is, ER_RW in this example.

3. Compile and link the C source:

```
armclang --target=arm-arm-eabi-none  -march=armv8-a file.c -g -c -O1 -o file.o
armlink --cpu=8-A.32 --scatter=scatter.scat --map file.o --output=file.axf
```

The --map option displays the memory map of the image.

In this example:

- __attribute__((section("foo"))) specifies that the linker is to place the global variable variable in a section called foo.
- __attribute__((section(".bss.mybss"))) specifies that the linker is to place the global variable z in a section called .bss.mybss.
- The scatter file specifies that the linker is to place the section foo in the ADDER execution region of the FLASH execution region.

The following example shows the output from --map:

```
…
    Execution Region ER_MYBSS (Base: 0x00008000, Size: 0x00000004, Max: 0x00002000,
ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00008000   0x00000004   Zero   RW            7    .bss.mybss          file.o
…
  Load Region FLASH (Base: 0x24000000, Size: 0x00000004, Max: 0x04000000, ABSOLUTE)

    Execution Region ADDER (Base: 0x08000000, Size: 0x00000004, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x08000000   0x00000004   Data   RW            5    foo                 file.o
…
```

Note

- If scatter-loading is not used, the linker places the section foo in the default ER_RW execution region of the LR_1 load region. It also places the section .bss.mybss in the default execution region ER_ZI.
- If you have a scatter file that does not include the foo selector, then the linker places the section in the defined RW execution region.

You can also place a function at a specific address using .ARM.__at_*address* as the section name. For example, to place the function sqr at 0x20000, specify:

```
int sqr(int n1) __attribute__((section(".ARM.__at_0x20000")));

int sqr(int n1)
{
    return n1*n1;
}
```

For more information, see *5.6 Placing functions and data at specific addresses* on page 5-71.

5.6 Placing functions and data at specific addresses

To place a single function or data item at a fixed address, you must enable the linker to process the function or data separately from the rest of the input files.

This section contains the following subsections:

5.6.1 Placing __at sections at a specific address

You can give a section a special name that encodes the address where it must be placed.

To place a section at a specific address, use the function or variable attribute `__attribute__((section("name")))` with the special name `.ARM.__at_address`.

To place ZI data at a specific address, use the variable attribute `__attribute__((section("name")))` with the special name `.bss.ARM.__at_address`

address is the required address of the section. The compiler normalizes this address to eight hexadecimal digits. You can specify the address in hexadecimal or decimal. Sections in the form of `.ARM.__at_address` are referred to by the abbreviation `__at`.

The following example shows how to assign a variable to a specific address in C or C++ code:

```
// place variable1 in a section called .ARM.__at_0x8000
int variable1 __attribute__((section(".ARM.__at_0x8000"))) = 10;
```

Note

The name of the section is only significant if you are trying to match the section by name in a scatter file. Without overlays, the linker automatically assigns `__at` sections when you use the `--autoat` command-line option. This option is the default. If you are using overlays, then you cannot use `--autoat` to place `__at` sections.

5.6.2 Restrictions on placing __at sections

There are restrictions when placing `__at` sections at specific addresses.

The following restrictions apply:

- `__at` section address ranges must not overlap, unless the overlapping sections are placed in different overlay regions.
- `__at` sections are not permitted in position independent execution regions.
- You must not reference the linker-defined symbols `$$Base`, `$$Limit` and `$$Length` of an `__at` section.
- `__at` sections must not be used in *Base Platform Application Binary Interface* (BPABI) executables and BPABI *dynamically linked libraries* (DLLs).
- `__at` sections must have an address that is a multiple of their alignment.
- `__at` sections ignore any `+FIRST` or `+LAST` ordering constraints.

5.6.3 Automatically placing __at sections

The linker automatically places `__at` sections, but you can override this feature.

The automatic placement of __at sections is enabled by default. Use the linker command-line option, --no_autoat to disable this feature.

Note

You cannot use __at section placement with position independent execution regions.

When linking with the --autoat option, the linker does not place __at sections with scatter-loading selectors. Instead, the linker places the __at section in a compatible region. If no compatible region is found, the linker creates a load and execution region for the __at section.

All linker execution regions created by --autoat have the UNINIT scatter-loading attribute. If you require a ZI __at section to be zero-initialized, then it must be placed within a compatible region. A linker execution region created by --autoat must have a base address that is at least 4 byte-aligned. If any region is incorrectly aligned, the linker produces an error message.

A compatible region is one where:

- The __at address lies within the execution region base and limit, where limit is the base address + maximum size of execution region. If no maximum size is set, the linker sets the limit for placing __at sections as the current size of the execution region without __at sections plus a constant. The default value of this constant is 10240 bytes, but you can change the value using the --max_er_extension command-line option.
- The execution region meets at least one of the following conditions:
  - It has a selector that matches the __at section by the standard scatter-loading rules.
  - It has at least one section of the same type (RO or RW) as the __at section.
  - It does not have the EMPTY attribute.

    Note

    The linker considers an __at section with type RW compatible with RO.

The following example shows the sections .ARM.__at_0x0000 type RO, .ARM.__at_0x4000 type RW, and .ARM.__at_0x8000 type RW:

```
// place the RO variable in a section called .ARM.__at_0x0000
const int foo __attribute__((section(".ARM.__at_0x0000"))) = 10;

// place the RW variable in a section called .ARM.__at_0x4000
int bar __attribute__((section(".ARM.__at_0x4000"))) = 100;

// place "variable" in a section called .ARM.__at_0x00008000
int variable __attribute__((section(".ARM.__at_0x00008000")));
```

The following scatter file shows how automatically to place these __at sections:

```
LR1 0x0
{
    ER_RO 0x0 0x4000
    {
        *(+RO)      ; .ARM.__at_0x0000 lies within the bounds of ER_RO
    }
    ER_RW 0x4000 0x2000
    {
        *(+RW)      ; .ARM.__at_0x4000 lies within the bounds of ER_RW
    }
    ER_ZI 0x6000 0x2000
    {
        *(+ZI)
    }
}
; The linker creates a load and execution region for the __at section
; .ARM.__at_0x8000 because it lies outside all candidate regions.
```

5.6.4 Manually placing __at sections

You can have direct control over the placement of __at sections, if required.

You can use the standard section-placement rules to place __at sections when using the --no_autoat command-line option.

Note

You cannot use __at section placement with position-independent execution regions.

The following example shows the placement of read-only sections .ARM.__at_0x2000 and the read-write section .ARM.__at_0x4000. Load and execution regions are not created automatically in manual mode. An error is produced if an __at section cannot be placed in an execution region.

The following example shows the placement of the variables in C or C++ code:

```
// place the RO variable in a section called .ARM.__at_0x2000
const int foo __attribute__((section(".ARM.__at_0x2000"))) = 100;
// place the RW variable in a section called .ARM.__at_0x4000
int bar __attribute__((section(".ARM.__at_0x4000")));
```

The following scatter file shows how to place __at sections manually:

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)               ; .ARM.__at_0x0000 is selected by +RO
    }
    ER_RO2 0x2000
    {
        *(.ARM.__at_0x02000)  ; .ARM.__at_0x2000 is selected by the section named
                              ; .ARM.__at_0x2000
    }
    ER2 0x4000
    {
        *(+RW, +ZI)           ; .ARM.__at_0x4000 is selected by +RW
    }
}
```

5.6.5 Placing a key in flash memory with an __at section

Some flash devices require a key to be written to an address to activate certain features. An __at section provides a simple method of writing a value to a specific address.

Placing the flash key variable in C or C++ code

Assume that a device has flash memory from 0x8000 to 0x10000 and a key is required in address 0x8000. To do this with an __at section, you must declare a variable so that the compiler can generate a section called .ARM.__at_0x8000.

```
// place flash_key in a section called .ARM.__at_0x8000
long flash_key __attribute__((section(".ARM.__at_0x8000")));
```

Manually placing a flash execution region

The following example shows how to manually place a flash execution region with a scatter file:

```
ER_FLASH 0x8000 0x2000
{
    *(+RW)
    *(.ARM.__at_0x8000) ; key
}
```

Use the linker command-line option --no_autoat to enable manual placement.

Automatically placing a flash execution region

The following example shows how to automatically place a flash execution region with a scatter file. Use the linker command-line option `--autoat` to enable automatic placement.

```
LR1 0x0
{
    ER_FLASH 0x8000 0x2000
    {
        *(+RO)                  ; other code and read-only data, the
                                ; __at section is automatically selected
    }
    ER2 0x4000
    {
        *(+RW +ZI)              ; Any other RW and ZI variables
    }
}
```

5.6.6 Placing constants at fixed locations

There are some situations when you want to place constants at fixed memory locations. For example, you might want to write a value to FLASH to read-protect a SoC device.

Procedure

1. Create a C file `abs_address.c` to define an integer and a string constant.

```
unsigned int const number = 0x12345678;
char* const string = "Hello World";
```

2. Create a scatter file, `scatter.scat`, to place the constants in separate sections `ER_RONUMBERS` and `ER_ROSTRINGS`.

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        *(+RO +RW)     ; all RO sections (must include section with
                       ; initial entry point)
    }
    ER_RONUMBERS +0
    {
        *(.rodata.number, +RO-DATA)
    }
    ER_ROSTRINGS +0
    {
        *(.rodata.string, .rodata.str1.1, +RO-DATA)
    }
                       ; rest of scatter-loading description

    ARM_LIB_STACK 0x80000 EMPTY -0x20000  ; Stack region growing down
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
}
```

`armclang` puts string literals in a section called `.rodata.str1.1`

3. Compile and link the file.

```
armclang --target=arm-arm-eabi-none -mcpu=cortex-a9 abs_address.c -g -c -o abs_address.o
armlink --cpu=cortex-a9 --scatter=scatter.scat abs_address.o --output=abs_address.axf
```

4. Run `fromelf` on the image to view the contents of the output sections.

```
fromelf -c -d abs_address.axf
```

The output contains the following sections:

```
...
** Section #2 'ER_RONUMBERS' (SHT_PROGBITS) [SHF_ALLOC]
    Size   : 4 bytes (alignment 4)
    Address: 0x00040000

    0x040000:   78 56 34 12                                      xV4.


** Section #3 'ER_ROSTRINGS' (SHT_PROGBITS) [SHF_ALLOC]
    Size   : 16 bytes (alignment 4)
```

```
    Address: 0x00040004

    0x040004:   48 65 6c 6c 6f 20 57 6f 72 6c 64 00 04 00 04 00    Hello World.....
...
```

5. Replace the `ER_RONUMBERS` and `ER_ROSTRINGS` sections in the scatter file with the following `ER_RODATA` section:

```
    ER_RODATA +0
    {
        abs_address.o(.rodata.number, .rodata.string, .rodata.str1.1, +RO-DATA)
    }
```

6. Repeat steps 3 and 4.
   The integer and string constants are both placed in the `ER_RODATA` section, for example:

```
** Section #2 'ER_RODATA' (SHT_PROGBITS) [SHF_ALLOC]
    Size   : 20 bytes (alignment 4)
    Address: 0x00040000

    0x040000:   78 56 34 12 48 65 6c 6c 6f 20 57 6f 72 6c 64 00    xV4.Hello World.
    0x040010:   04 00 04 00                                        ....
```

5.6.7 Placing jump tables in ROM

You might find that jump tables are placed in RAM rather than in ROM.

A jump table might be placed in a RAM `.data` section when you define it as follows:

```
typedef void PFUNC(void);
const PFUNC *table[3] = {func0, func1, func2};
```

The compiler also issues the warning:

```
jump.c:19:1: warning: 'const' qualifier on function type 'PFUNC'
      (aka 'void (void)') has unspecified behavior
const PFUNC *table[3] = {func0, func1, func2};
^~~~~
```

The following procedure describes how to place the jump table in a ROM `.rodata` section.

Procedure

1. Create a C file `jump.c`.

   Make the `PFUNC` type a pointer to a void function that has no parameters. You can then use `PFUNC` to create an array of constant function pointers.

```
extern void func0(void);
extern void func1(void);
extern void func2(void);

typedef void (*PFUNC)(void);

const PFUNC table[] = {func0, func1, func2};

void jump(unsigned i)
{
  if (i<=2)
  table[i]();
}
```

2. Compile the file.

```
armclang --target=arm-arm-eabi-none -mcpu=cortex-a9 jump.c -g -c -o jump.o
```

3. Run `fromelf` on the image to view the contents of the output sections.

```
fromelf -c -d jump.o
```

   The table is placed in the read-only section `.rodata` that you can place in ROM as required:

```
...
** Section #3 '.text.jump' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 64 bytes (alignment 4)
    Address: 0x00000000

    $a.0
```

```
    [Anonymous symbol #24]
    jump
        0x00000000:    e92d4800    .H-.    PUSH     {r11,lr}
        0x00000004:    e24dd008    ..M.    SUB      sp,sp,#8
        0x00000008:    e1a01000    ....    MOV      r1,r0
        0x0000000c:    e58d0004    ....    STR      r0,[sp,#4]
        0x00000010:    e3500002    ..P.    CMP      r0,#2
        0x00000014:    e58d1000    ....    STR      r1,[sp,#0]
        0x00000018:    8a000006    ....    BHI      {pc}+0x20 ; 0x38
        0x0000001c:    eaffffff    ....    B        {pc}+0x4 ; 0x20
        0x00000020:    e59d0004    ....    LDR      r0,[sp,#4]
        0x00000024:    e3001000    ....    MOVW     r1,#:LOWER16: table
        0x00000028:    e3401000    ..@.    MOVT     r1,#:UPPER16: table
        0x0000002c:    e7910100    ....    LDR      r0,[r1,r0,LSL #2]
        0x00000030:    e12fff30    0./.    BLX      r0
        0x00000034:    eaffffff    ....    B        {pc}+0x4 ; 0x38
        0x00000038:    e28dd008    ....    ADD      sp,sp,#8
        0x0000003c:    e8bd8800    ....    POP      {r11,pc}

...
** Section #7 '.rodata.table' (SHT_PROGBITS) [SHF_ALLOC]
    Size   : 12 bytes (alignment 4)
    Address: 0x00000000

    0x000000:   00 00 00 00 00 00 00 00 00 00 00 00               ............
...
```

5.6.8 Placing a variable at a specific address without scatter-loading

This example shows how to modify your source code to place code and data at specific addresses, and does not require a scatter file.

To place code and data at specific addresses without a scatter file:

1. Create the source file `main.c` containing the following code:

```
#include <stdio.h>

extern int sqr(int n1);
const int gValue __attribute__((section(".ARM.__at_0x5000"))) = 3; // Place at 0x5000
int main(void)
{
    int squared;
    squared=sqr(gValue);
    printf("Value squared is: %d\n", squared);
    return 0;
}
```

2. Create the source file `function.c` containing the following code:

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c function.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --map function.o main.o -o squared.axf
```

   The `--map` option displays the memory map of the image. Also, `--autoat` is the default.

In this example, `__attribute__((section(".ARM.__AT_0x5000")))` specifies that the global variable `gValue` is to be placed at the absolute address `0x5000`. `gValue` is placed in the execution region `ER$$.ARM.__AT_0x5000` and load region `LR$$.ARM.__AT_0x5000`.

The memory map shows:

```
…
  Load Region LR$$.ARM.__AT_0x5000 (Base: 0x00005000, Size: 0x00000004, Max: 0x00000004,
ABSOLUTE)

    Execution Region ER$$.ARM.__AT_0x5000 (Base: 0x00005000, Size: 0x00000004, Max:
0x00000004, ABSOLUTE, UNINIT)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00005000   0x00000004   Data   RO           18    .ARM.__AT_0x5000  main.o
```

5.6.9 Placing a variable at a specific address with scatter-loading

This example shows how to modify your source code to place code and data at a specific address using a scatter file.

To modify your source code to place code and data at a specific address using a scatter file:

1. Create the source file `main.c` containing the following code:

```
#include <stdio.h>
extern int sqr(int n1);
// Place at address 0x10000
const int gValue __attribute__((section(".ARM.__at_0x10000"))) = 3;
int main(void)
{
    int squared;
    squared=sqr(gValue);
    printf("Value squared is: %d\n", squared);
    return 0;
}
```

2. Create the source file `function.c` containing the following code:

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. Create the scatter file `scatter.scat` containing the following load region:

```
LR1 0x0
{
    ER1 0x0
    {
        *(+RO)                      ; rest of code and read-only data
    }
    ER2 +0
    {
        function.o
        *(.ARM.__at_0x10000)        ; Place gValue at 0x10000
    }
    ; RW and ZI data to be placed at 0x200000
    RAM 0x200000 (0x1FF00-0x2000)
    {
        *(+RW, +ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
```

   The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

4. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c function.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --no_autoat --scatter=scatter.scat --map function.o main.o -o squared.axf
```

   The `--map` option displays the memory map of the image.

The memory map shows that the variable is placed in the `ER2` execution region at address `0x10000`:

```
…
    Execution Region ER2 (Base: 0x00002a54, Size: 0x0000d5b0, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00002a54   0x0000001c   Code   RO            4    .text.sqr           function.o
    0x00002a70   0x0000d590   PAD
    0x00010000   0x00000004   Data   RO            9    .ARM.__at_0x10000   main.o
```

In this example, the size of `ER1` is unknown. Therefore, `gValue` might be placed in `ER1` or `ER2`. To make sure that `gValue` is placed in `ER2`, you must include the corresponding selector in `ER2` and link with the `--no_autoat` command-line option. If you omit `--no_autoat`, `gValue` is to placed in a separate load region `LR$$.ARM.__at_0x10000` that contains the execution region `ER$$.ARM.__at_0x10000`.

5.7 Placement of Arm® C and C++ library code

You can place code from the Arm standard C and C++ libraries using a scatter file.

Use `*armlib*` or `*libcxx*` so that the linker can resolve library naming in your scatter file.

Some Arm C and C++ library sections must be placed in a root region, for example `__main.o`, `__scatter*.o`, `__dc*.o`, and `*Region$$Table`. This list can change between releases. The linker can place all these sections automatically in a future-proof way with `InRoot$$Sections`.

Note

For AArch64, `__rtentry*.o` is moved to a root region.

This section contains the following subsections:

- *5.7.1 Placing code in a root region* on page 5-78.
- *5.7.2 Placing Arm® C library code* on page 5-78.
- *5.7.3 Placing Arm® C++ library code* on page 5-79.

5.7.1 Placing code in a root region

Some code must always be placed in a root region. You do this in a similar way to placing a named section.

To place all sections that must be in a root region, use the section selector `InRoot$$Sections`. For example :

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000      ; root region at 0x0
  {
    vectors.o (Vect, +FIRST)  ; Vector table
    * (InRoot$$Sections)      ; All library sections that must be in a
                              ; root region, for example, __main.o,
                              ; __scatter*.o, __dc*.o, and *Region$$Table
  }
  RAM 0x10000 0x8000
  {
    * (+RO, +RW, +ZI)         ; all other sections
  }
}
```

5.7.2 Placing Arm® C library code

You can place C library code using a scatter file.

To place C library code, specify the library path and library name as the module selector. You can use wildcard characters if required. For example:

```
LR1 0x0
{
    ROM1 0
    {
        * (InRoot$$Sections)
        * (+RO)
    }
    ROM2 0x1000
    {
        *armlib/c_* (+RO)                   ; all Arm-supplied C library functions
    }

    RAM1 0x3000
    {
        *armlib* (+RO)                      ; all other Arm-supplied library code
                                            ; for example, floating-point libraries
    }
    RAM2 0x4000
    {
        * (+RW, +ZI)
    }
}
```

The name `armlib` indicates the Arm C library files that are located in the directory *`install_directory`*`\lib\armlib`.

5.7.3 Placing Arm® C++ library code

You can place C++ library code using a scatter file.

To place C++ library code, specify the library path and library name as the module selector. You can use wildcard characters if required.

Procedure

1. Create the following C++ program, `foo.cpp`:

```
#include <iostream>

using namespace std;

extern "C" int foo ()
{
  cout << "Hello" << endl;
  return 1;
}
```

2. To place the C++ library code, define the following scatter file, `scatter.scat`:

```
LR 0x8000
{
    ER1 +0
    {
        *armlib*(+RO)
    }
    ER2 +0
    {
        *libcxx*(+RO)
    }
    ER3 +0
    {
        *(+RO)

        ; All .ARM.exidx* sections must be coalesced into a single contiguous
        ; .ARM.exidx section because the unwinder references linker-generated
        ; Base and Limit symbols for this section.
        *(0x70000001)  ; SHT_ARM_EXIDX sections

        ; All .init_array sections must be coalesced into a single contiguous
        ; .init_array section because the initialization code references
        ; linker-generated Base and Limit for this section.
        *(.init_array)
    }
    ER4 +0
    {
        *(+RW,+ZI)
    }
}
```

   The name `*armlib*` matches *`install_directory`*`\lib\armlib`, indicating the Arm C library files that are located in the `armlib` directory.

   The name `*libcxx*` matches *`install_directory`*`\lib\libcxx`, indicating the C++ library files that are located in the `libcxx` directory.

3. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c foo.cpp
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --scatter=scatter.scat --map main.o foo.o -o foo.axf
```

   The `--map` option displays the memory map of the image.

5.8 Placement of unassigned sections

The linker attempts to place input sections into specific execution regions. For any input sections that cannot be resolved, and where the placement of those sections is not important, you can specify where the linker is to place them.

To place sections that are not automatically assigned to specific execution regions, use the `.ANY` module selector in a scatter file.

Usually, a single `.ANY` selector is equivalent to using the `*` module selector. However, unlike `*`, you can specify `.ANY` in multiple execution regions.

The linker has default rules for placing unassigned sections when you specify multiple `.ANY` selectors. However, you can override the default rules using the following command-line options:

- `--any_contingency` to permit extra space in any execution regions containing `.ANY` sections for linker-generated content such as veneers and alignment padding.
- `--any_placement` to provide more control over the placement of unassigned sections.
- `--any_sort_order` to control the sort order of unassigned input sections.

In a scatter file, you can also:

- Assign a priority to a `.ANY` selector. This gives you more control over how the unassigned sections are divided between multiple execution regions. You can assign the same priority to more than one execution region.
- Specify the maximum size for an execution region that the linker can fill with unassigned sections.

This section contains the following subsections:

5.8.1 Default rules for placing unassigned sections

The linker has default rules for placing sections when using multiple `.ANY` selectors.

When more than one `.ANY` selector is present in a scatter file, the linker sorts sections in descending size order. It then takes the unassigned section with the largest size and assigns the section to the most specific `.ANY` execution region that has enough free space. For example, `.ANY(.text)` is judged to be more specific than `.ANY(+RO)`.

If several execution regions are equally specific, then the section is assigned to the execution region with the most available remaining space.

For example:

- You might have two equally specific execution regions where one has a size limit of `0x2000` and the other has no limit. In this case, all the sections are assigned to the second unbounded `.ANY` region.
- You might have two equally specific execution regions where one has a size limit of `0x2000` and the other has a size limit of `0x3000`. In this case, the first sections to be placed are assigned to the second `.ANY` region of size limit `0x3000`. This assignment continues until the remaining size of the second `.ANY` region is reduced to `0x2000`. From this point, sections are assigned alternately between both `.ANY` execution regions.

You can specify a maximum amount of space to use for unassigned sections with the execution region attribute `ANY_SIZE`.

5.8.2 Command-line options for controlling the placement of unassigned sections

You can modify how the linker places unassigned input sections when using multiple `.ANY` selectors by using a different placement algorithm or a different sort order.

The following command-line options are available:

- `--any_placement=`*`algorithm`*, where *`algorithm`* is one of `first_fit`, `worst_fit`, `best_fit`, or `next_fit`.
- `--any_sort_order=`*`order`*, where *`order`* is one of `cmdline` or `descending_size`.

Use `first_fit` when you want to fill regions in order.

Use `best_fit` when you want to fill regions to their maximum.

Use `worst_fit` when you want to fill regions evenly. With equal sized regions and sections `worst_fit` fills regions cyclically.

Use `next_fit` when you need a more deterministic fill pattern.

If the linker attempts to fill a region to its limit, as it does with `first_fit` and `best_fit`, it might overfill the region. This is because linker-generated content such as padding and veneers are not known until sections have been assigned to `.ANY` selectors. If this occurs you might see the following error:

`Error: L6220E: Execution region `*`regionname`*` size (`*`size`*` bytes) exceeds limit (`*`limit`*` bytes).`

The `--any_contingency` option prevents the linker from filling the region up to its maximum. It reserves a portion of the region's size for linker-generated content and fills this contingency area only if no other regions have space. It is enabled by default for the `first_fit` and `best_fit` algorithms, because they are most likely to exhibit this behavior.

5.8.3 Prioritizing the placement of unassigned sections

You can give a priority ordering when placing unassigned sections with multiple `.ANY` module selectors.

To prioritize the order of multiple `.ANY` sections use the `.ANY`*`num`* selector, where *`num`* is a positive integer starting at zero.

The highest priority is given to the selector with the highest integer.

The following example shows how to use `.ANY`*`num`*:

```
lr1 0x8000 1024
{
    er1 +0 512
    {
        .ANY1(+RO) ; evenly distributed with er3
    }
    er2 +0 256
    {
        .ANY2(+RO) ; Highest priority, so filled first
    }
    er3 +0 256
    {
        .ANY1(+RO) ; evenly distributed with er1
    }
}
```

5.8.4 Specify the maximum region size permitted for placing unassigned sections

You can specify the maximum size in a region that `armlink` can fill with unassigned sections.

Use the execution region attribute `ANY_SIZE` *`max_size`* to specify the maximum size in a region that `armlink` can fill with unassigned sections.

Be aware of the following restrictions when using this keyword:

- *`max_size`* must be less than or equal to the region size.
- If you use `ANY_SIZE` on a region without a `.ANY` selector, it is ignored by `armlink`.

When `ANY_SIZE` is present, `armlink` does not attempt to calculate contingency and strictly follows the `.ANY` priorities.

When ANY_SIZE is not present for an execution region containing a .ANY selector, and you specify the --any_contingency command-line option, then armlink attempts to adjust the contingency for that execution region. The aims are to:

- Never overflow a .ANY region.
- Make sure there is a contingency reserved space left in the given execution region. This space is reserved for veneers and section padding.

If you specify --any_contingency on the command line, it is ignored for regions that have ANY_SIZE specified. It is used as normal for regions that do not have ANY_SIZE specified.

Example

The following example shows how to use ANY_SIZE:

```
LOAD_REGION 0x0 0x3000
{
    ER_1 0x0 ANY_SIZE 0xF00 0x1000
    {
        .ANY
    }
    ER_2 0x0 ANY_SIZE 0xFB0 0x1000
    {
        .ANY
    }
    ER_3 0x0 ANY_SIZE 0x1000 0x1000
    {
        .ANY
    }
}
```

In this example:

- ER_1 has 0x100 reserved for linker-generated content.
- ER_2 has 0x50 reserved for linker-generated content. That is about the same as the automatic contingency of --any_contingency.
- ER_3 has no reserved space. Therefore, 100% of the region is filled, with no contingency for veneers. Omitting the ANY_SIZE parameter causes 98% of the region to be filled, with a two percent contingency for veneers.

5.8.5 Examples of using placement algorithms for .ANY sections

These examples show the operation of the placement algorithms for RO-CODE sections in sections.o.

The input section properties and ordering are shown in the following table:

Table 5-1 Input section properties for placement of .ANY sections

Name	Size
sec1	0x4
sec2	0x4
sec3	0x4
sec4	0x4
sec5	0x4
sec6	0x4

The scatter file used for the examples is:

```
LR 0x100
{
  ER_1 0x100 0x10
  {
     .ANY
  }
  ER_2 0x200 0x10
  {
```

```
    .ANY
  }
}
```

Note

These examples have `--any_contingency` disabled.

Example for first_fit, next_fit, and best_fit

This example shows the image memory map where several sections of equal size are assigned to two regions with one selector. The selectors are equally specific, equivalent to `.ANY(+R0)` and have no priority.

```
    Execution Region ER_1 (Base: 0x00000100, Size: 0x00000010, Max: 0x00000010, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000100   0x00000004   Code   RO            1    sec1               sections.o
    0x00000104   0x00000004   Code   RO            2    sec2               sections.o
    0x00000108   0x00000004   Code   RO            3    sec3               sections.o
    0x0000010c   0x00000004   Code   RO            4    sec4               sections.o


    Execution Region ER_2 (Base: 0x00000200, Size: 0x00000008, Max: 0x00000010, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000200   0x00000004   Code   RO            5    sec5               sections.o
    0x00000204   0x00000004   Code   RO            6    sec6               sections.o
```

In this example:

- For `first_fit` the linker first assigns all the sections it can to `ER_1`, then moves on to `ER_2` because that is the next available region.
- For `next_fit` the linker does the same as `first_fit`. However, when `ER_1` is full it is marked as `FULL` and is not considered again. In this example, `ER_1` is completely full. `ER_2` is then considered.
- For `best_fit` the linker assigns `sec1` to `ER_1`. It then has two regions of equal priority and specificity, but `ER_1` has less space remaining. Therefore, the linker assigns `sec2` to `ER_1`, and continues assigning sections until `ER_1` is full.

Example for worst_fit

This example shows the image memory map when using the `worst_fit` algorithm.

```
    Execution Region ER_1 (Base: 0x00000100, Size: 0x0000000c, Max: 0x00000010, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000100   0x00000004   Code   RO            1    sec1               sections.o
    0x00000104   0x00000004   Code   RO            3    sec3               sections.o
    0x00000108   0x00000004   Code   RO            5    sec5               sections.o


    Execution Region ER_2 (Base: 0x00000200, Size: 0x0000000c, Max: 0x00000010, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000200   0x00000004   Code   RO            2    sec2               sections.o
    0x00000204   0x00000004   Code   RO            4    sec4               sections.o
    0x00000208   0x00000004   Code   RO            6    sec6               sections.o
```

The linker first assigns `sec1` to `ER_1`. It then has two equally specific and priority regions. It assigns `sec2` to the one with the most free space, `ER_2` in this example. The regions now have the same amount of space remaining, so the linker assigns `sec3` to the first one that appears in the scatter file, that is `ER_1`.

Note

The behavior of `worst_fit` is the default behavior in this version of the linker, and it is the only algorithm available in earlier linker versions.

5.8.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority

This example shows the operation of the `next_fit` placement algorithm for `RO-CODE` sections in `sections.o`.

The input section properties and ordering are shown in the following table:

Table 5-2 Input section properties for placement of sections with next_fit

Name	Size
sec1	0x14
sec2	0x14
sec3	0x10
sec4	0x4
sec5	0x4
sec6	0x4

The scatter file used for the examples is:

```
LR 0x100
{
  ER_1 0x100 0x20
  {
     .ANY1(+RO-CODE)
  }
  ER_2 0x200 0x20
  {
     .ANY2(+RO)
  }
  ER_3 0x300 0x20
  {
     .ANY3(+RO)
  }
}
```

Note

This example has `--any_contingency` disabled.

The `next_fit` algorithm is different to the others in that it never revisits a region that is considered to be full. This example also shows the interaction between priority and specificity of selectors. This is the same for all the algorithms.

```
   Execution Region ER_1 (Base: 0x00000100, Size: 0x00000014, Max: 0x00000020, ABSOLUTE)

   Base Addr    Size         Type   Attr      Idx    E Section Name        Object

   0x00000100   0x00000014   Code   RO            1    sec1                sections.o


   Execution Region ER_2 (Base: 0x00000200, Size: 0x0000001c, Max: 0x00000020, ABSOLUTE)

   Base Addr    Size         Type   Attr      Idx    E Section Name        Object

   0x00000200   0x00000010   Code   RO            3    sec3                sections.o
   0x00000210   0x00000004   Code   RO            4    sec4                sections.o
   0x00000214   0x00000004   Code   RO            5    sec5                sections.o
   0x00000218   0x00000004   Code   RO            6    sec6                sections.o


   Execution Region ER_3 (Base: 0x00000300, Size: 0x00000014, Max: 0x00000020, ABSOLUTE)

   Base Addr    Size         Type   Attr      Idx    E Section Name        Object

   0x00000300   0x00000014   Code   RO            2    sec2                sections.o
```

In this example:

- The linker places `sec1` in `ER_1` because `ER_1` has the most specific selector. `ER_1` now has `0x6` bytes remaining.
- The linker then tries to place `sec2` in `ER_1`, because it has the most specific selector, but there is not enough space. Therefore, `ER_1` is marked as full and is not considered in subsequent placement steps. The linker chooses `ER_3` for `sec2` because it has higher priority than `ER_2`.
- The linker then tries to place `sec3` in `ER_3`. It does not fit, so `ER_3` is marked as full and the linker places `sec3` in `ER_2`.
- The linker now processes `sec4`. This is `0x4` bytes so it can fit in either `ER_1` or `ER_3`. Because both of these sections have previously been marked as full, they are not considered. The linker places all remaining sections in `ER_2`.
- If another section `sec7` of size `0x8` exists, and is processed after `sec6` the example fails to link. The algorithm does not attempt to place the section in `ER_1` or `ER_3` because they have previously been marked as full.

5.8.7 Examples of using sorting algorithms for .ANY sections

These examples show the operation of the sorting algorithms for `RO-CODE` sections in `sections_a.o` and `sections_b.o`.

The input section properties and ordering are shown in the following table:

Table 5-3 Input section properties and ordering for sections_a.o and sections_b.o

sections_a.o		sections_b.o	
Name	**Size**	**Name**	**Size**
seca_1	0x4	secb_1	0x4
seca_2	0x4	secb_2	0x4
seca_3	0x10	secb_3	0x10
seca_4	0x14	secb_4	0x14

Descending size example

The following linker command-line options are used for this example:

```
--any_sort_order=descending_size sections_a.o sections_b.o --scatter scatter.txt
```

The following table shows the order that the sections are processed by the `.ANY` assignment algorithm.

Table 5-4 Sort order for descending_size algorithm

Name	Size
seca_4	0x14
secb_4	0x14
seca_3	0x10
secb_3	0x10
seca_1	0x4
seca_2	0x4
secb_1	0x4
secb_2	0x4

With `--any_sort_order=descending_size`, sections of the same size use the creation index as a tiebreak.

Command-line example

The following linker command-line options are used for this example:

```
--any_sort_order=cmdline sections_a.o sections_b.o --scatter scatter.txt
```

The following table shows the order that the sections are processed by the `.ANY` assignment algorithm.

Table 5-5 Sort order for cmdline algorithm

Name	Size
seca_1	0x4
seca_2	0x4
seca_3	0x10
seca_4	0x14
secb_1	0x4
secb_2	0x4
secb_3	0x10
secb_4	0x14

That is, the input sections are sorted by command-line index.

5.8.8 Behavior when .ANY sections overflow because of linker-generated content

Because linker-generated content might cause `.ANY` sections to overflow, a contingency algorithm is included in the linker.

The linker does not know the address of a section until it is assigned to a region. Therefore, when filling `.ANY` regions, the linker cannot calculate the contingency space and cannot determine if calling functions require veneers. The linker provides a contingency algorithm that gives a worst-case estimate for padding and an additional two percent for veneers. To enable this algorithm use the `--any_contingency` command-line option.

The following diagram represents the notional image layout during `.ANY` placement:

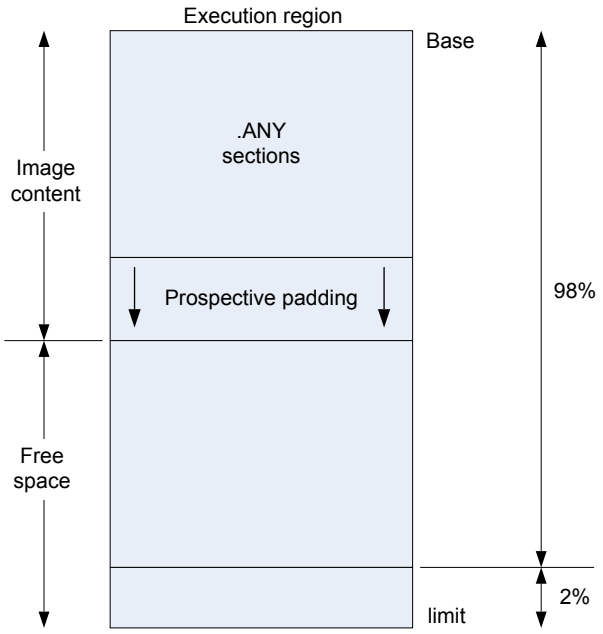


Figure 5-2 .ANY contingency

The downward arrows for prospective padding show that the prospective padding continues to grow as more sections are added to the `.ANY` selector.

Prospective padding is dealt with before the two percent veneer contingency.

When the prospective padding is cleared the priority is set to zero. When the two percent is cleared the priority is decremented again.

You can also use the `ANY_SIZE` keyword on an execution region to specify the maximum amount of space in the region to set aside for `.ANY` section assignments.

You can use the `armlink` command-line option `--info=any` to get extra information on where the linker has placed sections. This can be useful when trying to debug problems.

Example

1. Create the following `foo.c` program:

```
#include "stdio.h"

int array[10] __attribute__ ((section ("ARRAY")));

struct S {
    char A[8];
    char B[4];
};
struct S s;

struct S* get()
{
    return &s;
}

int sqr(int n1);
int gSquared __attribute__((section(".ARM.__at_0x5000")));  // Place at 0x5000

int sqr(int n1)
{
    return n1*n1;
}

int main(void) {
    int i;
    for (i=0; i<10; i++) {
        array[i]=i*i;
        printf("%d\n", array[i]);
    }
```

```
    gSquared=sqr(i);
    printf("%d squared is: %d\n", i, gSquared);

    return sizeof(array);
}
```

2. Create the following `scatter.scat` file:

```
LOAD_REGION 0x0 0x3000
{
   ER_1 0x0 0x1000
   {
      .ANY
   }
   ER_2 (ImageLimit(ER_1)) 0x1500
   {
      .ANY
   }
   ER_3 (ImageLimit(ER_2)) 0x500
   {
      .ANY
   }
   ER_4 (ImageLimit(ER_3)) 0x1000
   {
       *(+RW,+ZI)
   }
   ARM_LIB_STACK 0x800000 EMPTY -0x10000
   {
   }
   ARM_LIB_HEAP  +0 EMPTY 0x10000
   {
   }
}
```

3. Compile and link the program as follows:

```
armclang -c --target=arm-arm-none-eabi -mcpu=cortex-m4 -o foo.o foo.c
armlink --cpu=cortex-m4 --any_contingency --scatter=scatter.scat --info=any -o foo.axf
foo.o
```

The following shows an example of the information generated:

```
================================================================================


Sorting unassigned sections by descending size for .ANY placement.
Using Worst Fit .ANY placement algorithm.
.ANY contingency enabled.

Exec Region    Event                          Idx          Size        Section
Name                  Object
ER_2           Assignment: Worst fit          144
0x0000041a  .text                          c_wu.l(_printf_fp_dec.o)
ER_2           Assignment: Worst fit          261          0x00000338  CL$
$btod_div_common           c_wu.l(btod.o)
ER_1           Assignment: Worst fit          146
0x000002fc  .text                          c_wu.l(_printf_fp_hex.o)
ER_2           Assignment: Worst fit          260          0x00000244  CL$
$btod_mult_common          c_wu.l(btod.o)
...
ER_1           Assignment: Worst fit          3
0x00000090  .text                          foo.o
...
ER_3           Assignment: Worst fit          100          0x0000000a  .ARM.Collect$
$_printf_percent$$00000007  c_wu.l(_printf_ll.o)
ER_3           Info: .ANY limit reached       -            -
-                          -
ER_1           Assignment: Highest priority  423
0x0000000a  .text                          c_wu.l(defsig_exit.o)
...
.ANY contingency summary
Exec Region    Contingency     Type
ER_1           161             Auto
ER_2           180             Auto
ER_3           73              Auto

================================================================================


Sorting unassigned sections by descending size for .ANY placement.
Using Worst Fit .ANY placement algorithm.
.ANY contingency enabled.

Exec Region    Event                          Idx          Size        Section
```

```
Name                    Object
ER_2            Info: .ANY limit reached      -            -
-                             -
ER_1            Info: .ANY limit reached      -            -
-                             -
ER_3            Info: .ANY limit reached      -            -
-                             -
ER_2            Assignment: Worst fit         533          0x00000034  !!!
scatter                     c_wu.l(__scatter.o)
ER_2            Assignment: Worst fit         535          0x0000001c  !!
handler_zi                   c_wu.l(__scatter_zi.o)
```

5.9 Placing veneers with a scatter file

You can place veneers at a specific location with a linker-generated symbol.

Veneers allow switching between A32 and T32 code or allow a longer program jump than can be specified in a single instruction.

Procedure

1. To place veneers at a specific location, include the linker-generated symbol `Veneer$$Code` in a scatter file. At most, one execution region in the scatter file can have the `*(Veneer$$Code)` section selector.

   If it is safe to do so, the linker places veneer input sections into the region identified by the `*(Veneer$$Code)` section selector. It might not be possible for a veneer input section to be assigned to the region because of address range problems or execution region size limitations. If the veneer cannot be added to the specified region, it is added to the execution region containing the relocated input section that generated the veneer.

   Note

   Instances of `*(IWV$$Code)` in scatter files from earlier versions of Arm tools are automatically translated into `*(Veneer$$Code)`. Use `*(Veneer$$Code)` in new descriptions.

   `*(Veneer$$Code)` is ignored when the amount of code in an execution region exceeds 4MB of 16-bit T32 code, 16MB of 32-bit T32 code, and 32MB of A32 code.

   Note

   There are no state-change veneers in A64.

5.10 Preprocessing a scatter file

You can pass a scatter file through a C preprocessor. This permits access to all the features of the C preprocessor.

Use the first line in the scatter file to specify a preprocessor command that the linker invokes to process the file. The command is of the form:

```
#! preprocessor [pre_processor_flags]
```

Most typically the command is `#! armclang --target=arm-arm-none-eabi -march=armv8-a -E -x c`. This passes the scatter file through the `armclang` preprocessor.

You can:

- Add preprocessing directives to the top of the scatter file.
- Use simple expression evaluation in the scatter file.

For example, a scatter file, `file.scat`, might contain:

```
#! armclang --target=arm-arm-none-eabi -march=armv8-a -E -x c

#define ADDRESS 0x20000000
#include "include_file_1.h"

LR1 ADDRESS
{
    …
}
```

The linker parses the preprocessed scatter file and treats the directives as comments.

You can also use the `--predefine` command-line option to assign values to constants. For this example:

1. Modify `file.scat` to delete the directive `#define ADDRESS 0x20000000`.
2. Specify the command:

```
armlink --predefine="-DADDRESS=0x20000000" --scatter=file.scat
```

This section contains the following subsections:

- *5.10.1 Default behavior for armclang -E in a scatter file* on page 5-91.
- *5.10.2 Using other preprocessors in a scatter file* on page 5-91.

5.10.1 Default behavior for armclang -E in a scatter file

`armlink` behaves in the same way as `armclang` when invoking other Arm tools.

`armlink` searches for the `armclang` binary in the following order:

1. The same location as `armlink`.
2. The `PATH` locations.

`armlink` invokes `armclang` with the `-Iscatter_file_path` option so that any relative `#includes` work. The linker only adds this option if the full name of the preprocessor tool given is `armclang` or `armclang.exe`. This means that if an absolute path or a relative path is given, the linker does not give the `-Iscatter_file_path` option to the preprocessor. This also happens with the `--cpu` option.

On Windows, `.exe` suffixes are handled, so `armclang.exe` is considered the same as `armclang`. Executable names are case insensitive, so `ARMCLANG` is considered the same as `armclang`. The portable way to write scatter file preprocessing lines is to use correct capitalization and omit the `.exe` suffix.

5.10.2 Using other preprocessors in a scatter file

You must ensure that the preprocessing command line is appropriate for execution on the host system.

This means:

- The string must be correctly quoted for the host system. The portable way to do this is to use double-quotes.
- Single quotes and escaped characters are not supported and might not function correctly.
- The use of a double-quote character in a path name is not supported and might not work.

These rules also apply to any strings passed with the `--predefine` option.

All preprocessor executables must accept the `-o` *`file`* option to mean output to file and accept the input as a filename argument on the command line. These options are automatically added to the user command line by `armlink`. Any options to redirect preprocessing output in the user-specified command line are not supported.

5.11 Reserving an empty block of memory

You can reserve an empty block of memory with a scatter file, such as the area used for the stack.

To reserve an empty block of memory, add an execution region in the scatter file and assign the `EMPTY` attribute to that region.

This section contains the following subsections:

5.11.1 Characteristics of a reserved empty block of memory

An empty block of memory that is reserved with a scatter-loading description has certain characteristics.

The block of memory does not form part of the load region, but is assigned for use at execution time. Because it is created as a dummy ZI region, the linker uses the following symbols to access it:

- `Image$$`*`region_name`*`$$ZI$$Base`.
- `Image$$`*`region_name`*`$$ZI$$Limit`.
- `Image$$`*`region_name`*`$$ZI$$Length`.

If the length is given as a negative value, the address is taken to be the end address of the region. This address must be an absolute address and not a relative one.

5.11.2 Example of reserving an empty block of memory

This example shows how to reserve and empty block of memory for stack and heap using a scatter-loading description. It also shows the related symbols that the linker generates.

In the following example, the execution region definition `STACK 0x800000 EMPTY –0x10000` defines a region that is called `STACK`. The region starts at address `0x7F0000` and ends at address `0x800000`:

```
LR_1 0x80000                          ; load region starts at 0x80000
{
    STACK 0x800000 EMPTY -0x10000     ; region ends at 0x800000 because of the
                                      ; negative length. The start of the region
                                      ; is calculated using the length.
    {
                                      ; Empty region for placing the stack
    }

    HEAP +0 EMPTY 0x10000             ; region starts at the end of previous
                                      ; region. End of region calculated using
                                      ; positive length
    {
                                      ; Empty region for placing the heap
    }
    …                                 ; rest of scatter-loading description
}
```

Note

The dummy ZI region that is created for an `EMPTY` execution region is not initialized to zero at runtime.

If the address is in relative (`+`*`offset`*) form and the length is negative, the linker generates an error.

The following figure shows a diagrammatic representation for this example.

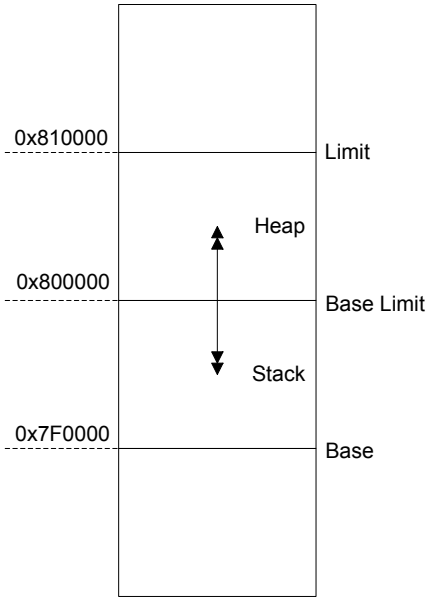


Figure 5-3 Reserving a region for the stack

In this example, the linker generates the following symbols:

```
Image$$STACK$$ZI$$Base      = 0x7f0000
Image$$STACK$$ZI$$Limit     = 0x800000
Image$$STACK$$ZI$$Length    = 0x10000
Image$$HEAP$$ZI$$Base       = 0x800000
Image$$HEAP$$ZI$$Limit      = 0x810000
Image$$HEAP$$ZI$$Length     = 0x10000
```

Note

The `EMPTY` attribute applies only to an execution region. The linker generates a warning and ignores an `EMPTY` attribute that is used in a load region definition.

The linker checks that the address space used for the `EMPTY` region does not coincide with any other execution region.

5.12 Aligning regions to page boundaries

You can produce an ELF file with each execution region starting at a page boundary.

The linker provides the following built-in functions to help create load and execution regions on page boundaries:

- `AlignExpr`, to specify an address expression.
- `GetPageSize`, to obtain the page size for use in `AlignExpr`. If you use `GetPageSize`, you must also use the `--paged` linker command-line option.
- `SizeOfHeaders()`, to return the size of the ELF header and Program Header table.

Note

- Alignment on an execution region causes both the load address and execution address to be aligned.
- The default page size is `0x8000`. To change the page size, specify the `--pagesize` linker command-line option.

To produce an ELF file with each execution region starting on a new page, and with code starting on the next page boundary after the header information:

```
LR1 0x0 + SizeOfHeaders()
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW AlignExpr(+0, GetPageSize())
    {
        *(+RW)
    }
    ER_ZI AlignExpr(+0, GetPageSize())
    {
        *(+ZI)
    }
}
```

If you set up your ELF file in this way, then you can memory-map it onto an operating system in such a way that:

- RO and RW data can be given different memory protections, because they are placed in separate pages.
- The load address everything expects to run at is related to its offset in the ELF file by specifying `SizeOfHeaders()` for the first load region.

5.13 Aligning execution regions and input sections

There are situations when you want to align code and data sections. How you deal with them depends on whether you have access to the source code.

Aligning when it is convenient for you to modify the source and recompile

When it is convenient for you to modify the original source code, you can align at compile time with the `__align(n)` keyword, for example.

Aligning when it is not convenient for you to modify the source and recompile

It might not be convenient for you to modify the source code for various reasons. For example, your build process might link the same object file into several images with different alignment requirements.

When it is not convenient for you to modify the source code, then you must use the following alignment specifiers in a scatter file:

`ALIGNALL`

Increases the section alignment of all the sections in an execution region, for example:

```
ER_DATA … ALIGNALL 8
{
    … ;selectors
}
```

`OVERALIGN`

Increases the alignment of a specific section, for example:

```
ER_DATA …
{
    *.o(.bar, OVERALIGN 8)
    … ;selectors
}
```

Chapter 6
Embedded Software Development

Describes how to develop embedded applications with Arm Compiler, with or without a target system present.

It contains the following sections:

6.1 About embedded software development

When developing embedded applications, the resources available in the development environment normally differ from those on the target hardware.

It is important to consider the process involved in moving an embedded application from the development or debugging environment to a system that runs standalone on target hardware.

When developing embedded software, you must consider the following:

- Understand the default compilation tool behavior and the target environment so that you appreciate the steps necessary to move from a debug or development build to a fully standalone production version of the application.
- Some C library functionality executes by using debug environment resources. If used, you must re-implement this functionality to make use of target hardware.
- The toolchain has no inherent knowledge of the memory map of any given target. You must tailor the image memory map to the memory layout of the target hardware.
- An embedded application must perform some initialization, such as stack and heap initialization, before the main application can be run. A complete initialization sequence requires code that you implement in addition to the Arm Compiler C library initialization routines.

6.2 Default compilation tool behavior

It is useful to be aware of the default behavior of the compilation tools if you do not yet know the full technical specifications of the target hardware.

For example, when you start work on software for an embedded application, you might not know the details of target peripheral devices, the memory map, or even the processor itself.

To enable you to proceed with software development before such details are known, the compilation tools have a default behavior that enables you to start building and debugging application code immediately.

In the Arm C library, support for some ISO C functionality, for example program I/O, can be provided by the host debugging environment. The mechanism that provides this functionality is known as semihosting. When semihosting is executed, the debug agent suspends program execution. The debug agent then uses the debug capabilities of the host (for example `printf` output to the debugger console) to service the semihosting operation before code execution is resumed on the target. The task performed by the host is transparent to the program running on the target.

Related information

What is semihosting?.

6.3 C library structure

Conceptually, the C library can be divided into functions that are part of the ISO C standard, for example `printf()`, and functions that provide support to the ISO C standard.

For example, the following figure shows the C library implementing the function `printf()` by writing to the debugger console window. This implementation is provided by calling `_sys_write()`, a support function that executes a semihosting call, resulting in the default behavior using the debugger instead of target peripherals.

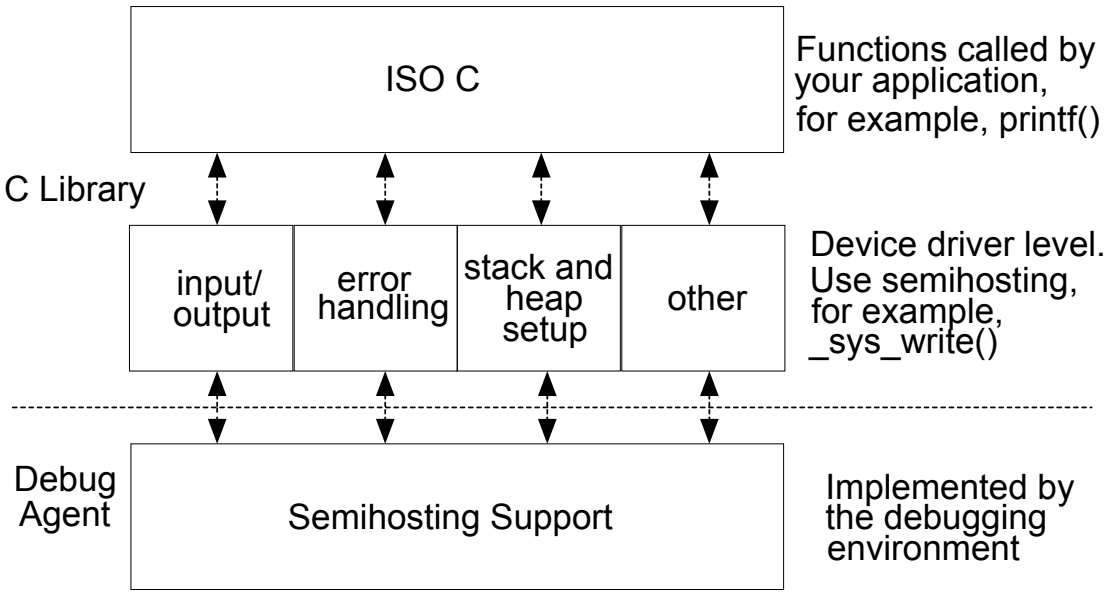


Figure 6-1 C library structure

Related information

The Arm C and C++ libraries.

The C and C++ library functions.

6.4 Default memory map

In an image where you have not described the memory map, the linker places code and data according to a default memory map.

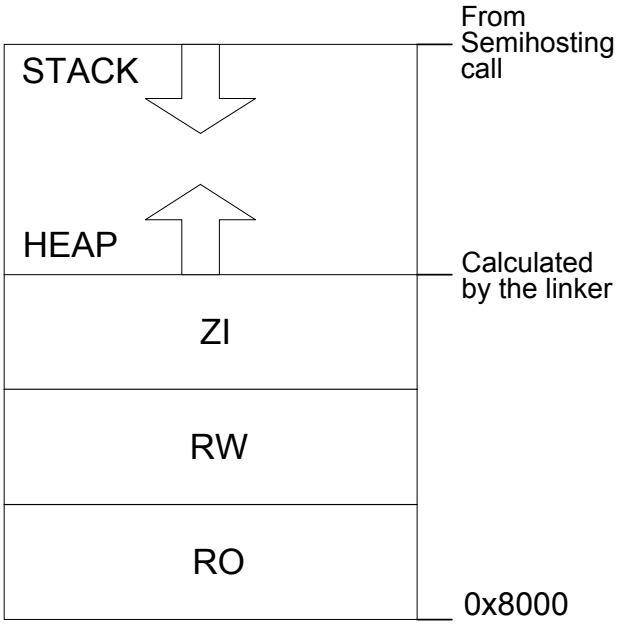


Figure 6-2 Default memory map

Note

The processors based on Armv6-M and Armv7-M architectures have fixed memory maps. This makes porting software easier between different systems based on these processors.

The default memory map is described as follows:

- The image is linked to load and run at address `0x8000`. All *Read Only* (RO) sections are placed first, followed by *Read-Write* (RW) sections, then *zero-initialized* (ZI) sections.
- The heap follows directly on from the top of ZI, so the exact location is decided at link time.
- The stack base location is provided by a semihosting operation during application startup. The value returned by this semihosting operation depends on the debug environment.

The linker observes a set of rules to decide where in memory code and data are located:

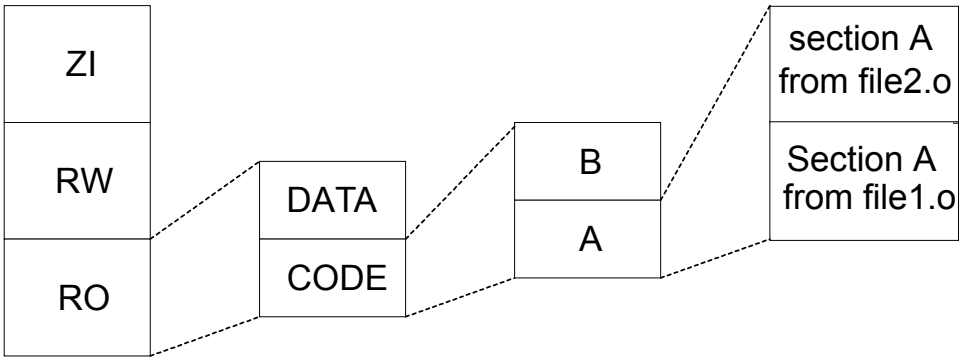


Figure 6-3 Linker placement rules

Generally, the linker sorts the input sections by attribute (RO, RW, ZI), by name, and then by position in the input list.

To fully control the placement of code and data you must use the scatter-loading mechanism.

Related concepts

6.6 Tailoring the C library to your target hardware on page 6-105.

Related information

The image structure.
Section placement with the linker.
About scatter-loading.
Scatter file syntax.
Cortex-M1 Technical Reference Manual.
Cortex-M3 Technical Reference Manual.

6.5 Application startup

In most embedded systems, an initialization sequence executes to set up the system before the main task is executed.

The following figure shows the default initialization sequence.

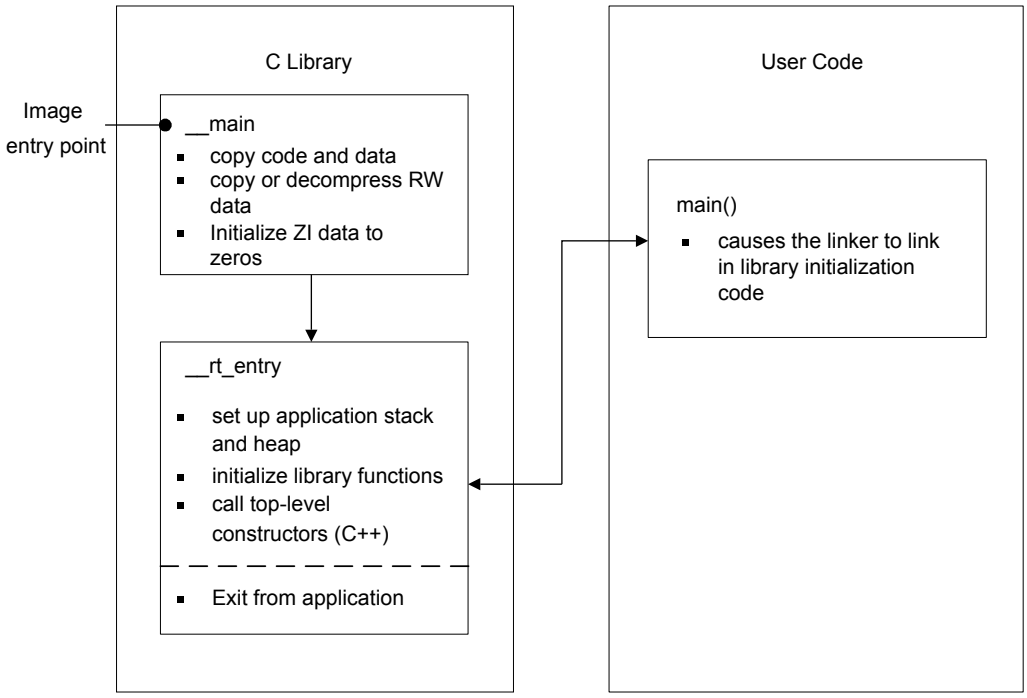


Figure 6-4 Default initialization sequence

`__main` is responsible for setting up the memory and `__rt_entry` is responsible for setting up the run-time environment.

`__main` performs code and data copying, decompression, and zero initialization of the ZI data. It then branches to `__rt_entry` to set up the stack and heap, initialize the library functions and static data, and call any top level C++ constructors. `__rt_entry` then branches to `main()`, the entry to your application. When the main application has finished executing, `__rt_entry` shuts down the library, then hands control back to the debugger.

The function label `main()` has a special significance. The presence of a `main()` function forces the linker to link in the initialization code in `__main` and `__rt_entry`. Without a function labeled `main()` the initialization sequence is not linked in, and as a result, some standard C library functionality is not supported.

Related information

--startup=symbol, --no_startup linker options.

Arm Compiler C Library Startup and Initialization.

6.6 Tailoring the C library to your target hardware

You can provide your own implementations of C library functions to override the default behavior.

By default, the C library uses semihosting to provide device driver level functionality, enabling a host computer to act as an input and an output device. This is useful because development hardware often does not have all the input and output facilities of the final system.

You can provide your own implementation of target-dependent C library functions to make use of target hardware. These are automatically linked in to your image in favor of the C library implementations. The following figure shows this process, known as retargeting the C library.

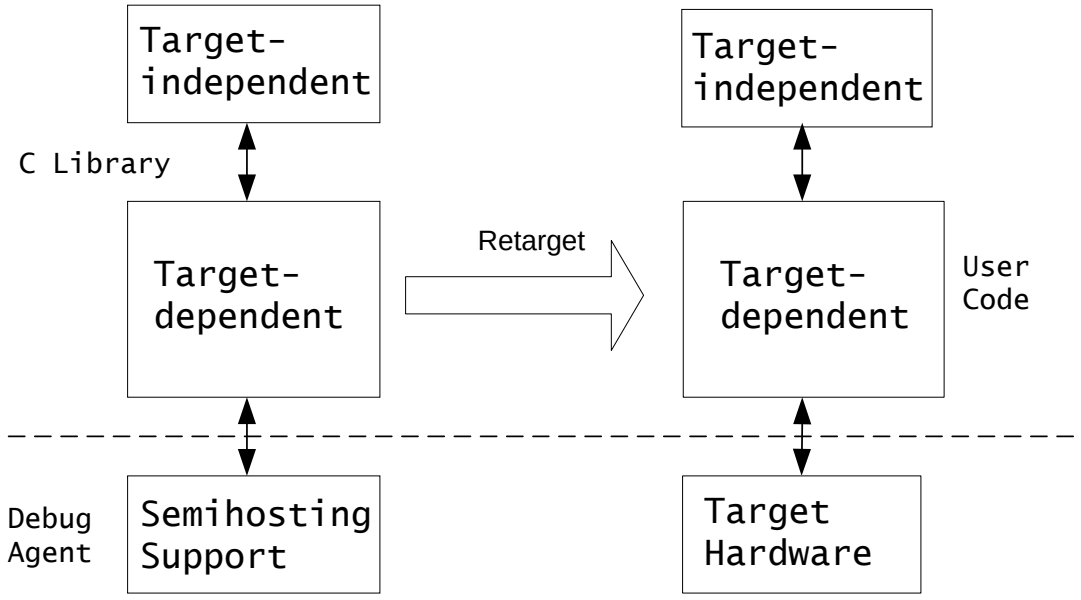


Figure 6-5 Retargeting the C library

For example, you might have a peripheral I/O device such as an LCD screen, and you might want to override the library implementation of `fputc()`, that writes to the debugger console, with one that outputs to the LCD. Because this implementation of `fputc()` is linked in to the final image, the entire `printf()` family of functions prints out to the LCD.

Example implementation of fputc()

In this example implementation of `fputc()`, the function redirects the input character parameter of `fputc()` to a serial output function `sendchar()` that is assumed to be implemented in a separate source file. In this way, `fputc()` acts as an abstraction layer between target dependent output and the C library standard output functions.

```
extern void sendchar(char *ch);
int fputc(int ch, FILE *f)
{   /* e.g. write a character to an LCD screen */
    char tempch = ch;
    sendchar(&tempch);
    return ch;
}
```

In a standalone application, you are unlikely to support semihosting operations. Therefore, you must remove all calls to target-dependent C library functions or re-implement them with non semihosting functions.

Related information

Using the libraries in a nonsemihosting environment.

6.7 Tailoring the image memory map to your target hardware

You can use a *scatter file* to define a memory map, giving you control over the placement of data and code in memory.

In your final embedded system, without semihosting functionality, you are unlikely to use the default memory map. Your target hardware usually has several memory devices located at different address ranges. To make the best use of these devices, you must have separate views of memory at load and run-time.

Scatter-loading enables you to describe the load and run-time memory locations of code and data in a textual description file known as a scatter file. This file is passed to the linker on the command line using the `--scatter` option. For example:

```
armlink --scatter scatter.scat file1.o file2.o
```

Scatter-loading defines two types of memory regions:

- Load regions containing application code and data at reset and load-time.
- Execution regions containing code and data when the application is executing. One or more execution regions are created from each load region during application startup.

A single code or data section can only be placed in a single execution region. It cannot be split.

During startup, the C library initialization code in `__main` carries out the necessary copying of code/data and zeroing of data to move from the image load view to the execute view.

Note

The overall layout of the memory maps of devices based around the Armv6-M and Armv7-M architectures are fixed. This makes it easier to port software between different systems based on these architectures.

Related information

Information about scatter files.
--scatter=filename linker option.
Armv7-M Architecture Reference Manual.
Armv6-M Architecture Reference Manual.

6.8 About the scatter-loading description syntax

In a scatter file, each region is defined by a header tag that contains, as a minimum, a name for the region and a start address. Optionally, you can add a maximum length and various attributes.

The scatter-loading description syntax shown in the following figure reflects the functionality provided by scatter-loading:

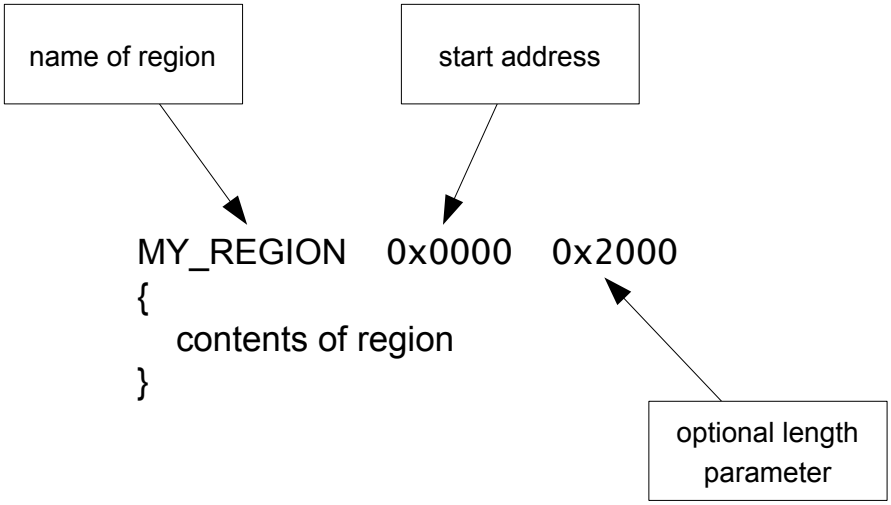


Figure 6-6 Scatter-loading description syntax

The contents of the region depend on the type of region:

- Load regions must contain at least one execution region. In practice, there are usually several execution regions for each load region.
- Execution regions must contain at least one code or data section, unless a region is declared with the `EMPTY` attribute. Non-`EMPTY` regions usually contain object or library code. You can use the wildcard (*) syntax to group all sections of a given attribute not specified elsewhere in the scatter file.

Related information

Information about scatter files.

Scatter-loading images with a simple memory map.

6.9 Root regions

A *root region* is an execution region with an execution address that is the same as its load address. A scatter file must have at least one root region.

One restriction placed on scatter-loading is that the code and data responsible for creating execution regions cannot be copied to another location. As a result, the following sections must be included in a root region:

- `__main.o` and `__scatter*.o` containing the code that copies code and data
- `__dc*.o` that performs decompression
- `Region$$Table` section containing the addresses of the code and data to be copied or decompressed.

Because these sections are defined as read-only, they are grouped by the `* (+RO)` wildcard syntax. As a result, if `* (+RO)` is specified in a non-root region, these sections must be explicitly declared in a root region using `InRoot$$Sections`.

Related information

About placing Arm C and C++ library code.

6.10 Placing the stack and heap

The scatter-loading mechanism provides a method for specifying the placement of the stack and heap in your image.

The application stack and heap are set up during C library initialization. You can tailor stack and heap placement by using the specially named `ARM_LIB_HEAP`, `ARM_LIB_STACK`, or `ARM_LIB_STACKHEAP` execution regions. Alternatively you can re-implement the `__user_setup_stackheap()` function if you are not using a scatter file.

Related concepts

6.11 Run-time memory models on page 6-111.

Related information

Tailoring the C library to a new execution environment.

Specifying stack and heap using the scatter file.

6.11 Run-time memory models

Arm Compiler toolchain provides one- and two-region run-time memory models.

One-region model

The application stack and heap grow towards each other in the same region of memory, see the following figure. In this run-time memory model, the heap is checked against the value of the stack pointer when new heap space is allocated, for example, when `malloc()` is called.

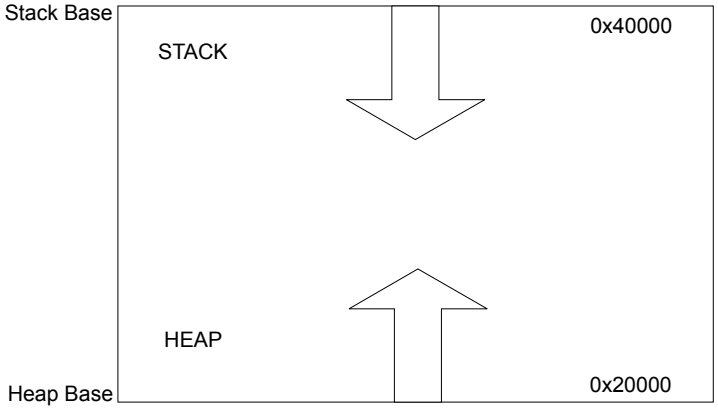


Figure 6-7 One-region model

One-region model routine

```
LOAD_FLASH ...
{
    ...
    ARM_LIB_STACKHEAP 0x20000 EMPTY 0x20000  ; Heap and stack growing towards
    { }                                      ; each other in the same region
    ...
}
```

Two-region model

The stack and heap are placed in separate regions of memory, see the following figure. For example, you might have a small block of fast RAM that you want to reserve for stack use only. For a two-region model you must import `__use_two_region_memory`.

In this run-time memory model, the heap is checked against the heap limit when new heap space is allocated.

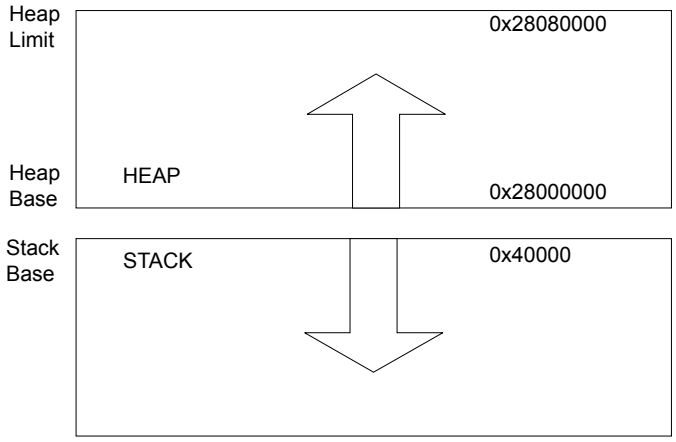


Figure 6-8 Two-region model

Two-region model routine

```
LOAD_FLASH ...
{
    ...
    ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
    { }                                   ;
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
    ...
}
```

In both run-time memory models, the stack grows unchecked.

6.12 Reset and initialization

The entry point to the C library initialization routine is `__main`. However, an embedded application on your target hardware performs some system-level initialization at startup.

Embedded system initialization sequence

The following figure shows a possible initialization sequence for an embedded system based on an Arm architecture:

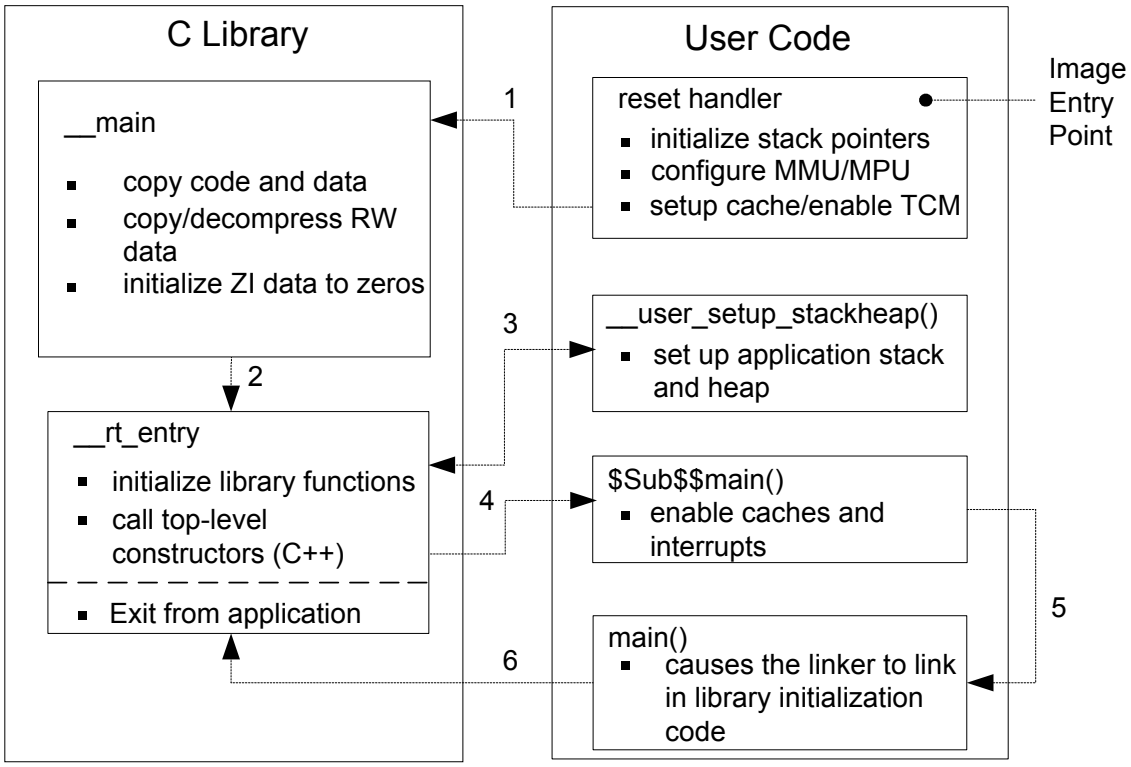


Figure 6-9 Initialization sequence

If you use a scatter file to tailor stack and heap placement, the linker includes a version of the library heap and stack setup code using the linker defined symbols, `ARM_LIB_*`, for these region names. Alternatively you can create your own implementation.

The reset handler is normally a short module coded in assembler that executes immediately on system startup. As a minimum, your reset handler initializes stack pointers for the modes that your application is running in. For processors with local memory systems, such as caches, TCMs, MMUs, and MPUs, some configuration must be done at this stage in the initialization process. After executing, the reset handler typically branches to `__main` to begin the C library initialization sequence.

There are some components of system initialization, for example, the enabling of interrupts, that are generally performed after the C library initialization code has finished executing. The block of code labeled `$Sub$$main()` performs these tasks immediately before the main application begins executing.

Related information

About using $Super$$ and $Sub$$ to patch symbol definitions.

Specifying stack and heap using the scatter file.

6.13 The vector table

All Arm systems have a vector table. It does not form part of the initialization sequence, but it must be present for an exception to be serviced.

It must be placed at a specific address, usually `0x0`. To do this you can use the scatter-loading `+FIRST` directive, as shown in the following example.

Placing the vector table at a specific address

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000      ; root region
  {
    vectors.o (Vect, +FIRST)  ; Vector table
    * (InRoot$$Sections)      ; All library sections that must be in a
                              ; root region, for example, __main.o,
                              ; __scatter*.o, __dc*.o, and * Region$$Table
  }
  RAM 0x10000 0x8000
  {
    * (+RO, +RW, +ZI)         ; all other sections
  }
}
```

The vector table for the microcontroller profiles is very different to most Arm architectures.

Related concepts

6.22 Vector table for ARMv6 and earlier, ARMv7-A and ARMv7-R profiles on page 6-123.
6.23 Vector table for M-profile architectures on page 6-124.

Related information

Information about scatter files.
Scatter-loading images with a simple memory map.

6.14 ROM and RAM remapping

You must consider what sort of memory your system has at address `0x0`, the address of the first instruction executed.

Note

This information does not apply to Armv6-M, Armv7-M, and Armv8-M profiles.

Note

This information assumes that an Arm processor begins fetching instructions at `0x0`. This is the standard behavior for systems based on Arm processors. However, some Arm processors, for example the processors based on the Armv7-A architecture, can be configured to begin fetching instructions from `0xFFFF0000`.

There has to be a valid instruction at `0x0` at startup, so you must have nonvolatile memory located at `0x0` at the moment of power-on reset. One way to achieve this is to have ROM located at `0x0`. However, there are some drawbacks to this configuration.

Example ROM/RAM remapping

This example shows a solution implementing ROM/RAM remapping after reset. The constants shown are specific to the Versatile board, but the same method is applicable to any platform that implements remapping in a similar way. Scatter files must describe the memory map after remapping.

```
;  System memory locations
Versatile_ctl_reg     EQU 0x101E0000 ; Address of control register
DEVCHIP_Remap_bit     EQU 0x100      ; Bit 8 is remap bit of control register
    ENTRY
; Code execution starts here on reset
; On reset, an alias of ROM is at 0x0, so jump to 'real' ROM.
        LDR     pc, =Instruct_2
Instruct_2
; Remap by setting remap bit of the control register
; Clear the DEVCHIP_Remap_bit by writing 1 to bit 8 of the control register
        LDR     R1, =Versatile_ctl_reg
        LDR     R0, [R1]
        ORR     R0, R0, #DEVCHIP_Remap_bit
        STR     R0, [R1]
; RAM is now at 0x0.
; The exception vectors must be copied from ROM to RAM
; The copying is done later by the C library code inside __main
; Reset_Handler follows on from here
```

6.15 Local memory setup considerations

Many Arm processors have on-chip memory management systems, such as Memory Management Units (MMU) or Memory Protection Units (MPU). These devices are normally set up and enabled during system startup.

Therefore, the initialization sequence of processors with local memory systems requires special consideration.

The C library initialization code in `__main` is responsible for setting up the execution time memory map of the image. Therefore, the run-time memory view of the processor must be set up before branching to `__main`. This means that any MMU or MPU must be set up and enabled in the reset handler.

Tightly Coupled Memories (TCM) must also be enabled before branching to `__main`, normally before MMU/MPU setup, because you generally want to scatter-load code and data into TCMs. You must be careful that you do not have to access memory that is masked by the TCMs when they are enabled.

You might also encounter problems with cache coherency if caches are enabled before branching to `__main`. Code in `__main` copies code regions from their load address to their execution address, essentially treating instructions as data. As a result, some instructions can be cached in the data cache, in which case they are not visible to the instruction path.

To avoid these coherency problems, enable caches after the C library initialization sequence finishes executing.

Related information

Cortex-A Series Programmer's Guide for Armv8-A.
Cortex-A Series Programmer's Guide for Armv7-A.
Cortex-R Series Programmer's Guide for Armv7-R.

6.16 Stack pointer initialization

As a minimum, your reset handler must assign initial values to the stack pointers of any execution modes that are used by your application.

Example stack pointer initialization

In this example, the stacks are located at `stack_base`:

```
; *****************************************************************
; This example does not apply to Armv6-M and Armv7-M profiles
; *****************************************************************
Len_FIQ_Stack    EQU     256
Len_IRQ_Stack    EQU     256
stack_base      DCD      0x18000
;
Reset_Handler
    ; stack_base could be defined above, or located in a scatter file
    LDR     R0, stack_base ;
    ; Enter each mode in turn and set up the stack pointer
    MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit    ; Interrupts disabled
    MOV     sp, R0
    SUB     R0, R0, #Len_FIQ_Stack
    MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit    ; Interrupts disabled
    MOV     sp, R0
    SUB     R0, R0, #Len_IRQ_Stack
    MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit    ; Interrupts disabled
    MOV     sp, R0
    ; Leave processor in SVC mode
```

The `stack_base` symbol can be a hard-coded address, or it can be defined in a separate assembler source file and located by a scatter file.

The example allocates 256 bytes of stack for *Fast Interrupt Request* (FIQ) and *Interrupt Request* (IRQ) mode, but you can do the same for any other execution mode. To set up the stack pointers, enter each mode with interrupts disabled, and assign the appropriate value to the stack pointer.

The stack pointer value set up in the reset handler is automatically passed as a parameter to `__user_initial_stackheap()` by C library initialization code. Therefore, this value must not be modified by `__user_initial_stackheap()`.

Related information

Specifying stack and heap using the scatter file.
Cortex-M3 Embedded Software Development.

6.17 Hardware initialization

In general, it is beneficial to separate all system initialization code from the main application. However, some components of system initialization, for example, enabling of caches and interrupts, must occur after executing C library initialization code.

Use of $Sub and $Super

You can make use of the `$Sub` and `$Super` function wrapper symbols to insert a routine that is executed immediately before entering the main application. This mechanism enables you to extend functions without altering the source code.

This example shows how `$Sub` and `$Super` can be used in this way:

```
extern void $Super$$main(void);
void $Sub$$main(void)
{
    cache_enable();    // enables caches
    int_enable();      // enables interrupts
    $Super$$main();    // calls original main()
}
```

The linker replaces the function call to `main()` with a call to `$Sub$$main()`. From there you can call a routine that enables caches and another to enable interrupts.

The code branches to the real `main()` by calling `$Super$$main()`.

Related information

About using $Super$$ and $Sub$$ to patch symbol definitions.

6.18 Execution mode considerations

You must consider the mode in which the main application is to run. Your choice affects how you implement system initialization.

Note

This does not apply to Armv6-M, Armv7-M, and Armv8-M profiles.

Much of the functionality that you are likely to implement at startup, both in the reset handler and `$Sub$$main`, can only be done while executing in privileged modes, for example, on-chip memory manipulation, and enabling interrupts.

If you want to run your application in a privileged mode, this is not an issue. Ensure that you change to the appropriate mode before exiting your reset handler.

If you want to run your application in User mode, however, you can only change to User mode after completing the necessary tasks in a privileged mode. The most likely place to do this is in `$Sub$$main()`.

Note

The C library initialization code must use the same stack as the application. If you need to use a non-User mode in `$Sub$$main` and User mode in the application, you must exit your reset handler in System mode, which uses the User mode stack pointer.

6.19 Target hardware and the memory map

It is better to keep all information about the memory map of a target, including the location of target hardware peripherals and the stack and heap limits, in your scatter file, rather than hard-coded in source or header files.

Mapping to a peripheral register

Conventionally, addresses of peripheral registers are hard-coded in project source or header files. You can also declare structures that map on to peripheral registers, and place these structures in the scatter file.

For example, if a target has a timer peripheral with two memory mapped 32-bit registers, a C structure that maps to these registers is:

```
struct
{
    volatile unsigned ctrl;           /* timer control */
    volatile unsigned tmr;            /* timer value   */
} timer_regs;
```

Note

You can also use `__attribute__((section(".ARM.__at_address")))` to specify the absolute address of a variable.

Placing the mapped structure

To place this structure at a specific address in the memory map, you can create an execution region containing the module that defines the structure. The following example shows an execution region called `TIMER` that locates the `timer_regs` structure at `0x40000000`:

```
ROM_LOAD 0x24000000 0x04000000
{
; ...
    TIMER 0x40000000 UNINIT
    {
        timer_regs.o (+ZI)
    }
    ; ...
}
```

It is important that the contents of these registers are not zero-initialized during application startup, because this is likely to change the state of your system. Marking an execution region with the `UNINIT` attribute prevents ZI data in that region from being zero-initialized by `__main`.

Related tasks

5.6 Placing functions and data at specific addresses on page 5-71.

Related information

__attribute__((section("name"))) variable attribute.

6.20 Execute-only memory

Execute-only memory (XOM) allows only instruction fetches. Read and write accesses are not allowed.

Execute-only memory allows you to protect your intellectual property by preventing executable code being read by users. For example, you can place firmware in execute-only memory and load user code and drivers separately. Placing the firmware in execute-only memory prevents users from trivially reading the code.

Note

The Arm architecture does not directly support execute-only memory. Execute-only memory is supported at the memory device level.

Related tasks

6.21 Building applications for execute-only memory on page 6-122.

6.21 Building applications for execute-only memory

Placing code in execute-only memory prevents users from trivially reading that code.

Note

Link Time Optimization does not honor the `armclang -mexecute-only` option. If you use the `armclang -flto` or `-Omax` options, then the compiler cannot generate execute-only code and produces a warning.

To build an application with code in execute-only memory:

Procedure

1. Compile your C or C++ code using the `-mexecute-only` option.
   ```
   armclang --target=arm-arm-none-eabi -march=armv7-m -mexecute-only -c test.c -o test.o
   ```
   The `-mexecute-only` option prevents the compiler from generating any data accesses to the code sections.

   To keep code and data in separate sections, the compiler disables the placement of literal pools inline with code.

   Compiled execute-only code sections in the ELF object file are marked with the `SHF_ARM_NOREAD` flag.
2. Specify the memory map to the linker using either of the following:
   - The `+XO` selector in a scatter file.
   - The `armlink --xo-base` option on the command-line.

   ```
   armlink --xo-base=0x8000 test.o -o test.axf
   ```
   The XO execution region is placed in a separate load region from the RO, RW, and ZI execution regions.

   Note

   If you do not specify `--xo-base`, then by default:
   - The XO execution region is placed immediately before the RO execution region, at address `0x8000`.
   - All execution regions are in the same load region.

Related concepts

6.20 Execute-only memory on page 6-121.

Related information

-mexecute-only compiler option.
--execute_only assembler option.
--xo_base=address linker option.
AREA.

6.22 Vector table for ARMv6 and earlier, ARMv7-A and ARMv7-R profiles

The vector table for Armv6 and earlier, Armv7-A and Armv7-R profiles consists of branch or load PC instructions to the relevant handlers.

If required, you can include the FIQ handler at the end of the vector table to ensure it is handled as efficiently as possible, see the following example. Using a literal pool means that addresses can easily be modified later if necessary.

Typical vector table using a literal pool

```
                AREA vectors, CODE, READONLY
                ENTRY
Vector_Table
                LDR pc, Reset_Addr
                LDR pc, Undefined_Addr
                LDR pc, SVC_Addr
                LDR pc, Prefetch_Addr
                LDR pc, Abort_Addr
                NOP                     ;Reserved vector
                LDR pc, IRQ_Addr
FIQ_Handler
                ; FIQ handler code - max 4kB in size
Reset_Addr      DCD Reset_Handler
Undefined_Addr  DCD Undefined_Handler
SVC_Addr        DCD SVC_Handler
Prefetch_Addr   DCD Prefetch_Handler
Abort_Addr      DCD Abort_Handler
IRQ_Addr        DCD IRQ_Handler
                ...
                END
```

This example assumes that you have ROM at location `0x0` on reset. Alternatively, you can use the scatter-loading mechanism to define the load and execution address of the vector table. In that case, the C library copies the vector table for you.

Note

The vector table for Armv6 and earlier architectures supports A32 instructions only. Armv6T2 and later architectures support both T32 instructions and A32 instructions in the vector table. This does not apply to the Armv6-M, Armv7-M, and Armv8-M profiles.

6.23 Vector table for M-profile architectures

The vector table for the microcontroller profiles consists of addresses to the relevant handlers.

The handler for exception number *n* is held at (*vectorbaseaddress* + 4 * *n*).

In Armv7-M and Armv8-M processors you can specify the *vectorbaseaddress* in the *Vector Table Offset Register* (VTOR) to relocate the vector table. The default location on reset is `0x0` (CODE space). For Armv6-M, the vector table base address is fixed at `0x0`. The word at *vectorbaseaddress* holds the reset value of the main stack pointer.

Note

The least significant bit, bit[0] of each address in the vector table must be set or a HardFault exception is generated. Arm Compiler toolchain normally enables this for you if T32 symbol names are used in the table.

6.24 Vector Table Offset Register

In Armv7-M and Armv8-M, the Vector Table Offset Register locates the vector table in CODE, RAM, or SRAM space.

When setting a different location, the offset, in bytes, must be aligned to:

- a power of 2.
- a minimum of 128 bytes.
- a minimum of 4*N, where N is the number of exceptions supported.

The minimal alignment is 128 bytes, which allows for 32 exceptions. 16 registers are reserved for system exceptions, and therefore, you can use for up to 16 interrupts.

To use more interrupts, you must adjust the alignment by rounding up to the next power of two. For example, if you require 21 interrupts, then the total number of exceptions is 37 (21 plus 16 reserved system exceptions). The alignment must be on a 64-word boundary because the next power of 2 after 37 is 64.

——— **Note** ———

Implementations might restrict where the vector table can be located. For example, in Cortex-M3 r0p0 to r2p0, the vector table cannot be in RAM space.

———————————

Appendix A
Supporting reference information

The various features in Arm Compiler might have different levels of support, ranging from fully supported product features to community features.

It contains the following sections:

A.1 Support level definitions

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

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

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

Identification in the documentation

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

Product features

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

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

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

Beta product features

Beta product features are implementation complete, but have not been sufficiently tested to be regarded as suitable for use in production environments.
Beta product features are indicated with [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 will remain functional across update releases, although changes are expected to be unlikely.

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

Guidance on use of community features

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

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

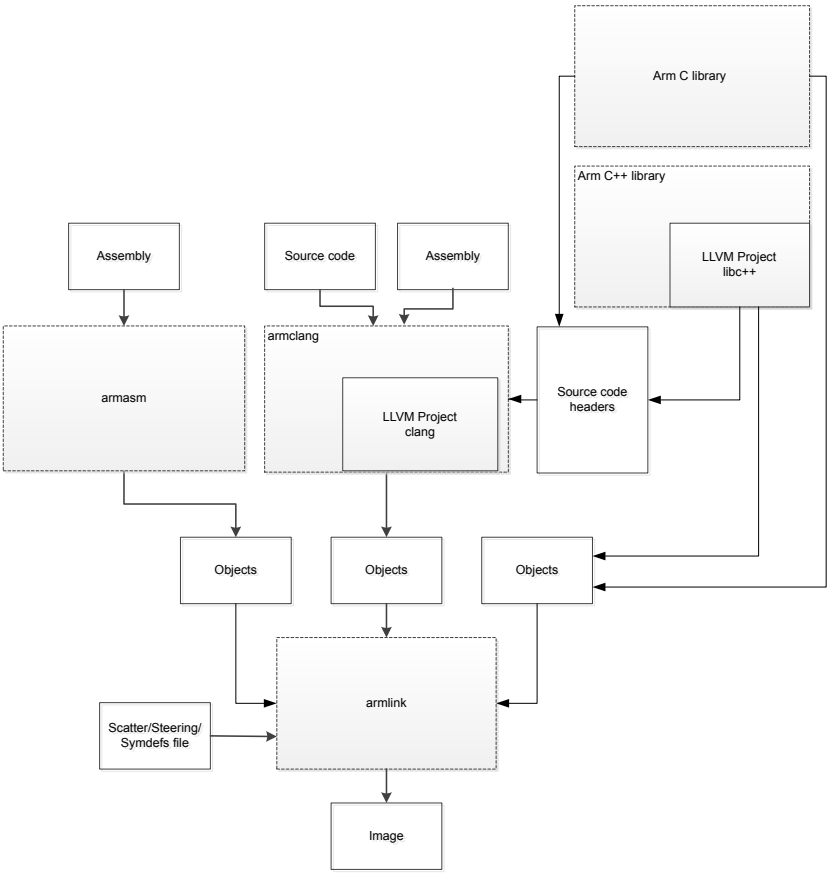


Figure A-1 Integration boundaries in Arm Compiler 6.

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 Appx-A-127.

List of known unsupported features

The following is an incomplete list of unsupported features, and might change over time:
- The Clang option `-stdlib=libstdc++` is not supported.
- C++ static initialization of local variables is not thread-safe when linked against the standard C++ libraries. For thread-safety, you must provide your own implementation of thread-safe functions as described in *Standard C++ library implementation definition*.

  ——— **Note** ———

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

  ———————————

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

A.2 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.

A.3 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 64-bit Arm 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.

A.4 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.

A.5 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*.

To set an environment variable on a Windows machine:

1. Open the **System** settings from the Control Panel.
2. Click **Advanced system settings** to display the System Properties dialog box, then click **Environment Variables...**.
3. Create a new user variable for the required environment variable.

To set an environment variable on a Linux machine, using a `bash` shell, use the `export` command on the command-line. For example:

```
export ARM_TOOL_VARIANT=ult
```

Table A-1 Environment variables used by the toolchain

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

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

Environment variable	Setting
`ARMCOMPILER6_LINKOPT`	An optional environment variable to define additional linker options that are to be used outside your regular makefile. The options listed appear before any options specified for the `armlink` command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.
`ARMROOT`	Your installation directory root, *`install_directory`*.
`ARMLMD_LICENSE_FILE`	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. **Note** On Windows, the length of `ARMLMD_LICENSE_FILE` must not exceed 260 characters.
`C_INCLUDE_PATH`	GCC-compatible environment variable. Adds the specified directories to the list of places that are searched to find included C files.
`COMPILER_PATH`	GCC-compatible environment variable. Adds the specified directories to the list of places that are searched to find subprograms.
`CPATH`	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.
`CPLUS_INCLUDE_PATH`	GCC-compatible environment variable. Adds the specified directories to the list of places that are searched to find included C++ files.
`TMP`	Used on Windows platforms to specify the directory to be used for temporary files.
`TMPDIR`	Used on Red Hat Linux platforms to specify the directory to be used for temporary files.

A.6 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.

A.7 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.