Building your first embedded image

Version 1.0

Non-Confidential

Issue 03

102432_0100_03_en

Building your first embedded image

Release information

Document history

Issue	Date	Confidentiality	Change
0100-03	16 June 2019	Non-Confidential	Initial release

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.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

Coding for an embedded system typically requires the programmer to have more direct interaction with the device hardware than a programmer writing software for a general-purpose computer. This is because:

- Embedded systems may not have a display, so the programmer might need to retarget debug output to a serial output port.
- Embedded systems typically monitor inputs waiting for a stimulus, and these events will require an interrupt handler.
- Embedded systems often require low-level initialization at startup, before any other code is executed, in the form of a reset handler.

This guide shows you how to write, compile, and run a simple program for an embedded system based on Arm technology. This information is useful for anyone who is new to writing software for an Arm-based embedded system.

This guide is the first in a collection of related documentation:

- Building your first embedded image (this guide)
- Retargeting output to UART
- Creating an event-driven embedded image
- Changing Exception level and Security state in an embedded image

At the end of this guide, you will be able to:

- Write a simple Hello World example program for an embedded system
- Configure the memory map
- Build an Executable and Linkable Format (ELF) image using the Arm Compiler 6 toolchain
- Run a simulation of the ELF image on the supplied FVP model

2. Before you begin

To complete this guide, you will need to have Arm Development Studio Gold Edition installed. If you do not have Arm Development Studio, you can download a 30-day free trial.

Arm Development Studio Gold Edition is a professional quality tool chain developed by Arm to accelerate your first steps in Arm software development. It includes both the Arm Compiler 6 toolchain and the FVP_Base_Cortex-A73x2-A53x4 model that you will use in this guide. We will use the command-line tools for most of the guide, which means that you will need to configure your environment in order to run Arm Compiler 6 from the command-line.

The individual sections of this guide contain some code examples. These code examples are available to download as a ZIP file:

- CommonTasks-BuildingYourFirstEmbeddedImage.zip

If you want to use the Arm Development Studio GUI instead of the command line tools, follow the instructions in Arm Development Studio Getting Started Guide, Tutorial: Hello World.

3. Write and compile Hello World

Let's start with a simple C program, and use the armclang and armlink tools to compile and generate an executable image.

1. In your command-line terminal, use your favorite editor, for example, vi, to create a new file called `hello_world.c` with the following contents:

```
#include <stdio.h>

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

2. Compile the C code to object code with armclang:

```
$ armclang -c -g --target=aarch64-arm-none-eabi hello_world.c
```

   This command tells the armclang compiler to compile `hello_world.c` for the Armv8-A architecture and generate an ELF object file `hello_world.o`. The options used in this command are:

   - `-c` tells the compiler to stop after compiling to object code. We will perform the link step to create the final executable in the next step.
   - `-g` tells the compiler to include debug information in the image.
   - `--target=aarch64-arm-none-eabi` tells the compiler to target the Armv8-A AArch64 ABI.

3. Create an executable image by linking the object using armlink. This generates an ELF image file named `__image.axf`:

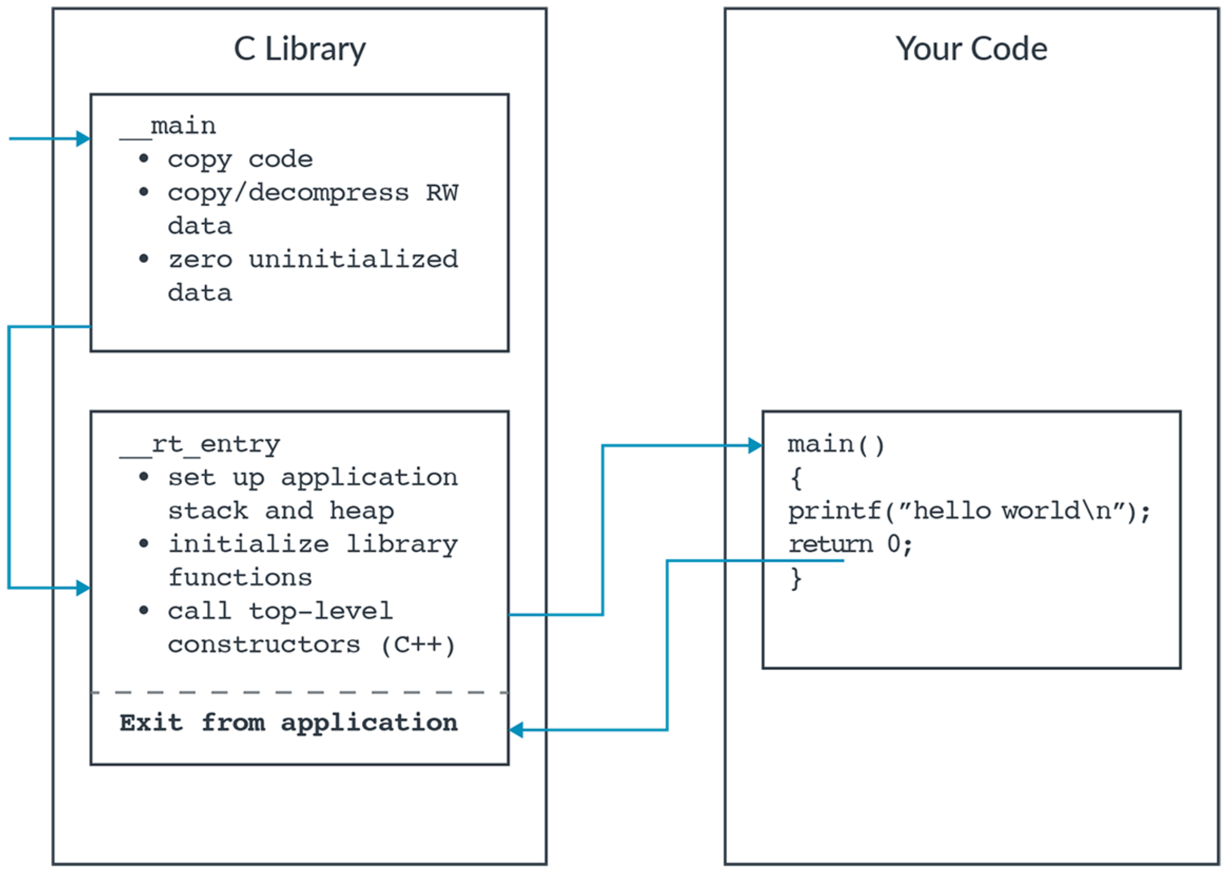
```
$ armlink hello_world.o
```

   Because we have not specified an entry point, when you run this image the entry point defaults to `__main()` in the Arm libraries. These libraries perform a number of setup activities, including:

   - Copying all the code and data from the image into memory.
   - Setting up an area of memory for the application stack and heap.
   - Branching to the `main()` function to run the application.

This diagram illustrates the code startup sequence that shows how control passes from the C library to your code:

Figure 3-1: Write and compile hello world diagram.



4. Specify the memory map

If you tried to execute the image that you created in the last step on the FVP_Base_Cortex-A73x2-A53x4 model, it would not run. This is because the default memory map used by armlink does not match the memory map used by the model. Instead of using the default memory map, you will specify a new memory map that matches the model and allows the image to run successfully. To do this, you will create a scatter file that tells the linker the structure of the memory map.

The memory map describes the different regions of target memory, and what they can be used for. For example, ROM can hold read-only code and data but cannot store read-write data.

You can see the memory map for the model in the following diagram:

Figure 4-1: Specify the memory map diagram.

Create a scatter file to tell the linker about the structure of the memory map:

1. Create a new file scatter.txt in the same directory as hello_world.c with the following contents:

```
ROM_LOAD 0x00000000 0x00010000
  {
    ROM_EXEC +0x0
    {
      * (+RO)
    }

    RAM_EXEC 0x04000000 0x10000
    {
      * (+RW, +ZI)
    }
    ARM_LIB_STACKHEAP 0x04010000 ALIGN 64 EMPTY 0x10000
    {}
  }
```

2. Rebuild the image using the scatter file:

```
$ armclang -c -g --target=aarch64-arm-none-eabi hello_world.c
$ armlink --scatter=scatter.txt hello_world.o
```

Advanced information

The statements in the scatter file define the different regions of memory and their purpose.

Let's look at them sequentially. The following instruction defines a load region.

```
ROM_LOAD 0x00000000 0x00010000
  {...}
```

A load region is an area of memory that contains the image file at reset before execution starts. The first number specified gives the starting address of the region, and the second number gives the size of the region.

The following instruction defines an execution region:

```
ROM_EXEC +0x0
  {
    * (+RO)
  }
```

Execution regions define the memory locations in which different parts of the image will be placed at run-time.

An execution region is called a root region if it has the same load-time and execute-time address. `ROM_EXEC` qualifies as a root region because its execute-time is located at an offset of `+0x0` from the start of the load region (that is, the region has the same load-time and execute-time addresses).

The initial entry point of an image must be in a root region. In our scatter file, all read-only (RO) code including the entry point `__main()` is placed in the `ROM_EXEC` root region.

```
RAM_EXEC 0x04000000 0x10000
    {
      * (+RW, +ZI)
    }
```

`RAM_EXEC` contains any read-write (RW) or zero-initialised (ZI) data. Because this has been placed in SRAM, it is not a root region.

This instruction specifies the placement of the heap and stack:

```
ARM_LIB_STACKHEAP 0x04010000 EMPTY 0x10000
    {}
```

- The heap starts at `0x04010000` and grows upward.
- The stack starts at `0x0401FFFF` and grows downwards.

The `EMPTY` declaration reserves `0x10000` of uninitialized memory, starting at `0x04010000`.

`ARM_LIB_STACKHEAP` and `EMPTY` are syntactically significant for the linker. However, `ROM_LOAD`, `ROM_EXEC`, and `RAM_EXEC` are not syntactically significant and could be renamed if you like.

For more information about memory maps, scatter files, and armlink, please refer to the armlink documentation.

5. Run the image with a model

You can now run the executable image `__image.axf` from the command line using the FVP_Base_Cortex-A73x2-A53x4 model:

```
$ FVP_Base_Cortex-A73x2-A53x4 __image.axf -C pctl.startup=0.0.1.0
```

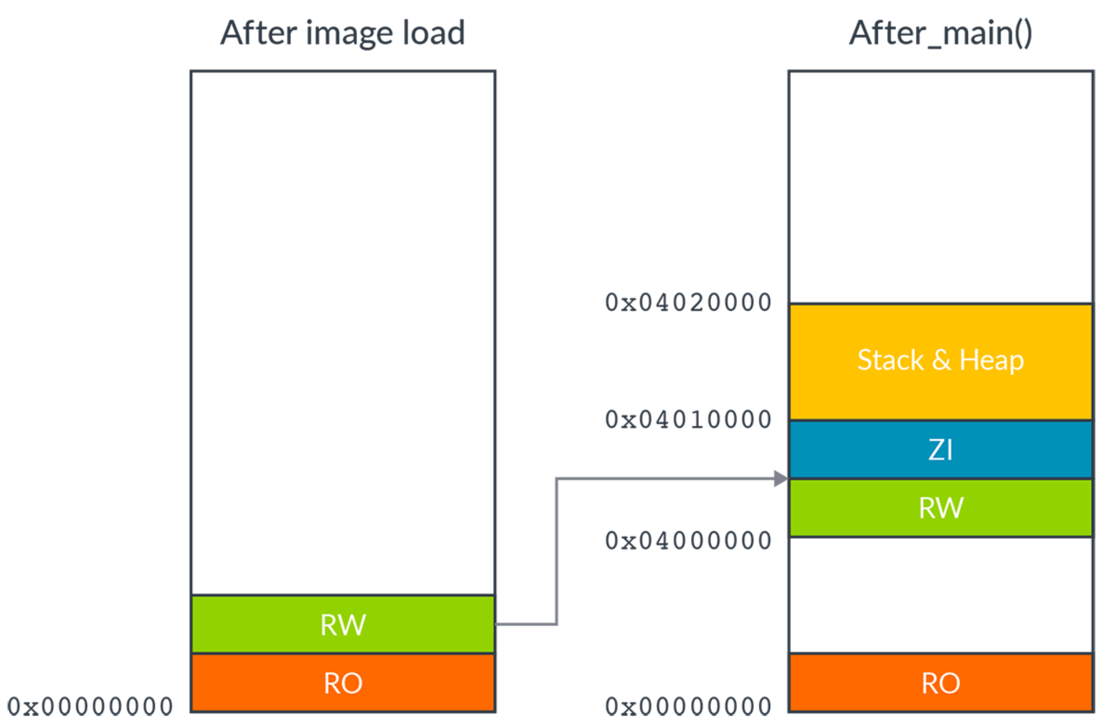
When the model is running, the message hello world appears on your screen.

By default, the model boots up multiple cores. This could lead to strange or inconsistent behaviors, such as multiple hello world prints. To avoid this type of result, we use the `-c pctl.startup=0.0.1.0` option to specify that only a single core should be used.

Another method to avoid strange or inconsistent results is to write some startup code that shuts down all but one core. We will discuss writing startup code later in this guide.

At reset, the code and data will be in the `ROM_LOAD` section. The library function `__main()` is responsible for copying the RW and ZI data, and `__rt_entry()` sets up the stack and heap. The Arm documentation, for example the Arm Compiler armlink User Guide, refers to this process as scatter loading.

Figure 5-1: Run the image with a model diagram.

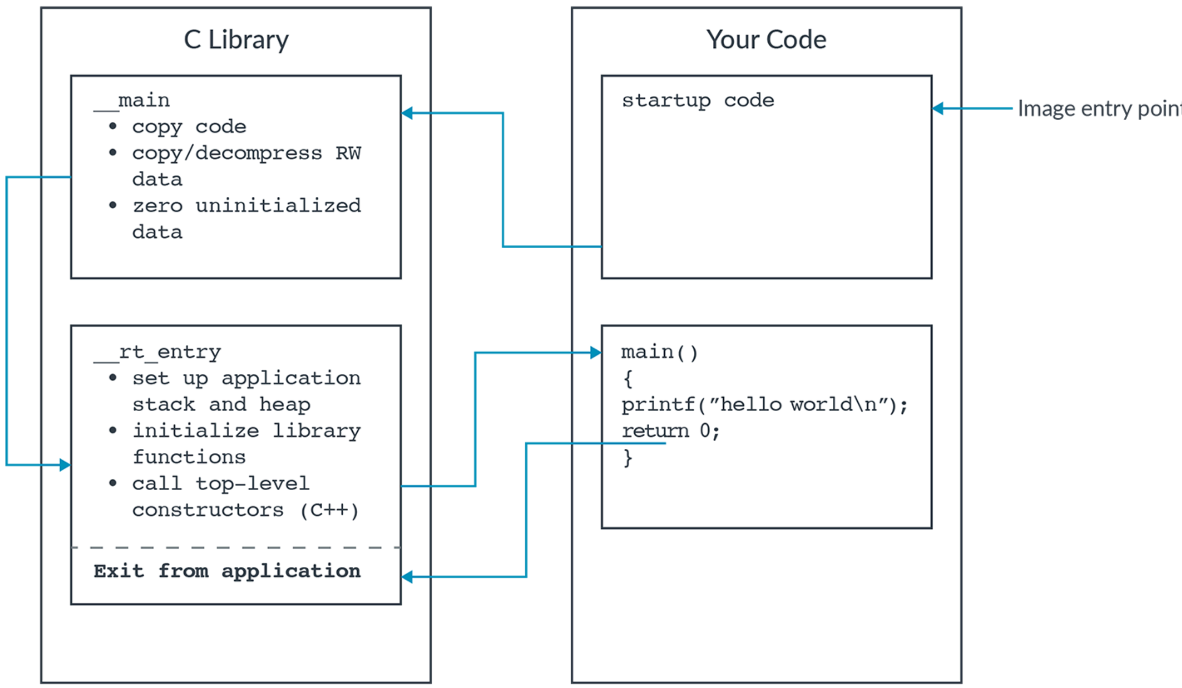


6. Write a reset handler

Typically, an embedded system needs some low-level initialization at startup.

Often this initialization must occur before any other code is executed. This means that you must define and change the entry point for the system in a way that reflects the execution flow that is shown in the following diagram:

Figure 6-1: Write a reset handler diagram.



1. Create a new file, `startup.s`, with the following contents:

```
  .section  BOOT,"ax" // Define an executable ELF section, BOOT
  .align 3                     // Align to 2^3 byte boundary

  .global start64
  .type start64, "function"
start64:


  // Which core am I
  // ----------------
  MRS      x0, MPIDR_EL1
  AND      x0, x0, #0xFFFF     // Mask off to leave Aff0 and Aff1
  CBZ      x0, boot            // If not *.*.0.0, then go to sleep
sleep:
  WFI
  B        sleep

boot:
  // Disable trapping of CPTR_EL3 accesses or use of Adv.SIMD/FPU
  // -------------------------------------------------------------
```

```
  MSR      CPTR_EL3, xzr       // Clear all trap bits

  // Branch to scatter loading and C library init code
  .global  __main
  B        __main
```

The `MPIDR_EL1` register provides a CPU identification mechanism. The `Aff0` and `Aff1` bitfields let us check which numbered CPU in a cluster the code is running on. This startup code sends all but one CPU to sleep. The status of the Floating Point Unit (FPU) in the model is unknown. The Architectural Feature Trap Register, `CPTR_EL3`, has no defined reset value. Setting `CPTR_EL3` to zero disables trapping of SIMD, FPU, and a few other instructions.

2. Compile the startup code:

```
$ armclang -c -g --target=aarch64-arm-none-eabi startup.s
```

3. Modify the scatter file so that the startup code goes into the root region `ROM_EXEC`:

```
ROM_EXEC +0x0
  {
    startup.o(BOOT, +FIRST)
    * (+RO)
  }
```

Adding the line `startup.o(BOOT, +FIRST)` ensures that the `BOOT` section of our startup file is placed first in the `ROM_EXEC` region.

4. Link the objects, specifying an entry label for the linker. Execution branches to this entry label on reset:

```
$ armlink --scatter=scatter.txt --entry=start64 hello_world.o startup.o
```

5. Run the executable image `__image.axf` from the command-line:

```
$ FVP_Base_Cortex-A73x2-A53x4 __image.axf
```

The message hello world appears on your screen.

7. Related information

Here are some resources related to material in this guide:

- Arm Community (ask development questions, and find articles and blogs on specific topics from Arm experts)
- Arm Compiler 6 documentation set
- Arm Compiler 6 - Bare-metal Hello World C using the Armv8 model (blog using the DS-5 GUI)
- Fixed Virtual Platforms (FVP) Documentation
- Placing the stack and heap with a scatter file
- The scatter-loading mechanism (for information about scatter files)
- Armv8-a Learn the Architecture series of guides

Useful links for training:

- Introduction to Armv8-A
- Memory model overview

8. Next steps

This guide is the first in a series of four guides on the topic of building an embedded image. In this guide, you learned how to write and run an embedded program, how to write and compile Hello World!, to specify the memory map, to run the image with a model, and to write a reset handler.

You can continue learning about building an embedded image in the next guides in the series:

- Retarget embedded output to UART
- Create an event-driven embedded image
- Changing exception level and security state in an embedded image