Embedded Programming with the GNU Toolchain

Embedded Programming with the GNU Toolchain

Vijay Kumar B.

<vijaykumar@bravegnu.org>
Revision History

Table of Contents

1. Introduction
2. Setting up the ARM Lab

    2.1. Qemu ARM
    2.2. Installing Qemu in Debian
    2.3. Installing GNU Toolchain for ARM

3. Hello ARM

    3.1. Building the Binary
    3.2. Executing in Qemu
    3.3. More Monitor Commands

4. More Assembler Directives

    4.1. Sum an Array
    4.2. String Length

5. Using RAM
6. Linker

    6.1. Symbol Resolution
    6.2. Relocation

7. Linker Script File

    7.1. Linker Script Example

8. Data in RAM, Example

    8.1. RAM is Volatile!
    8.2. Specifying Load Address
    8.3. Copying `.data` to RAM

9. Exception Handling
10. C Startup

    10.1. Stack

10.2. Global Variables
10.3. Read-only Data
10.4. Startup Code
11. Using the C Library
12. Inline Assembly
13. Contributing
14. Credits

14.1. People
14.2. Tools

15. Tutorial Copyright
A. ARM Programmer's Model
B. ARM Instruction Set
C. ARM Stacks

Workshop Alert!

The author of this tutorial is doing a workshop on "ARM Bare Metal Programming", in Chennai, India. If you are interested in the workshop, please visit https://in.explara.com/e/arm-bare-metal-programming, to book your ticket. To get notified about future workshops, follow us on Twitter @zilogic.

1. Introduction

The GNU toolchain is increasingly being used for deeply embedded software development. This type of software development is also called standalone C programming and bare metal C programming. Standalone C programming brings along with it new problems, and dealing with them requires a deeper understanding of the GNU toolchain. The GNU toolchain's manuals provide excellent information on the toolchain, but from the perspective of the toolchain, rather than the perspective of the problem. Well, that is how manuals are supposed to be written anyway. The result is that the answers to common problems are scattered all over, and new users of the GNU toolchain are left baffled.

This tutorial attempts to bridge the gap by explaining the tools from the perspective of the problem. Hopefully, this should enable more people to use the GNU toolchain for their embedded projects.

For the purpose of this tutorial, an ARM based embedded system is emulated using Qemu. With this you can learn the GNU toolchain from the comforts of your desktop, without having to invest on hardware. This tutorial itself does not teach the ARM instruction set. It is supposed to be used with other books and on-line tutorials like:

- ARM Assembler - http://www.heyrick.co.uk/assembler/
- ARM Assembly Language Programming - http://www.arm.com/miscPDFs/9658.pdf

But for the convenience of the reader, frequently used ARM instructions are listed in the appendix.

2. Setting up the ARM Lab


2. Setting up the ARM Lab

This section shows how to setup a simple ARM development and testing environment in your PC, using Qemu and the GNU toolchain. Qemu is a machine emulator capable of emulating various machines including ARM based machines. You can write ARM assembly programs, compile them using the GNU toolchain and execute and test them in Qemu.

2.1. Qemu ARM

Qemu will be used to emulate a PXA255 based *connex* board from Gumstix. You should have at least version 0.9.1 of Qemu to work with this tutorial.

The PXA255 has an ARM core with a ARMv5TE compliant instruction set. The PXA255 also has several on-chip peripherals. Some peripherals will be introduced in the course of the tutorial.

2.2. Installing Qemu in Debian

This tutorial requires qemu version 0.9.1 or above. The qemu package available in Debian Squeeze/Wheezy, meets this requirement. Install `qemu` using `apt-get`.

```
$ apt-get install qemu
```

2.3. Installing GNU Toolchain for ARM

1. Folks at CodeSourcery (part of Mentor Graphics) have been kind enough to make GNU toolchains available for various architectures. Download the GNU toolchain for ARM, available from from http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/
2. Extract the tar archive, to `~/toolchains`.

   ```
   $ mkdir ~/toolchains
   $ cd ~/toolchains
   $ tar -jxf ~/downloads/arm-2008q1-126-arm-none-eabi-i686-pc-linux-gnu.
   ```

3. Add the toolchain to your `PATH`.

   ```
   $ PATH=$HOME/toolchains/arm-2008q1/bin:$PATH
   ```

4. You might want to add the previous line to your `.bashrc`.


Embedded Programming with the GNU Toolchain

3. Hello ARM


3. Hello ARM

In this section, you will learn to assemble a simple ARM program, and test it on a bare metal connex board emulated by Qemu.

The assembly program source file consists of a sequence of statements, one per line. Each statement has the following format.

```
label:    instruction          @ comment
```

Each of the components is optional.

`label`
: The label is a convenient way to refer to the location of the instruction in memory. The label can be used where ever an address can appear, for example as an operand of the branch instruction. The label name should consist of alphabets, digits, `_` and `$`.

`comment`
: A comment starts with an `@`, and the characters that appear after an `@` are ignored.

`instruction`
: The `instruction` could be an ARM instruction or an assembler directive. Assembler directives are commands to the assembler. Assembler directives always start with a `.` (period).

Here is a very simple ARM assembly program to add two numbers.

Listing 1. Adding Two Numbers

```
        .text
start:                          @ Label, not really required
        mov    r0, #5           @ Load register r0 with the value 5
        mov    r1, #4           @ Load register r1 with the value 4
        add    r2, r1, r0       @ Add r0 and r1 and store in r2

stop:   b stop                  @ Infinite loop to stop execution
```

The `.text` is an assembler directive, which says that the following instructions have to be assembled into the code section, rather than the `.data` section. Sections will be covered in detail, later in the tutorial.

3.1. Building the Binary

Save the program in a file say `add.s`. To assemble the file, invoke the GNU Toolchain's assembler `as`, as shown in the following command.

```
$ arm-none-eabi-as -o add.o add.s
```

The `-o` option specifies the output filename.

Note
Cross toolchains are always prefixed with the target architecture for which they are built, to avoid name conflicts with the host toolchain. For the sake readability, tools will be referred to without the prefix, in the text.

To generate the executable file, invoke the GNU Toolchain's linker `ld`, as shown in the following command.

```
$ arm-none-eabi-ld -Ttext=0x0 -o add.elf add.o
```

Here again, the `-o` option specifies the output filename. The `-Ttext=0x0`, specifies that addresses should be assigned to the labels, such that the instructions were starting from address `0x0`. To view the address assignment for various labels, the `nm` command can be used as shown below.

```
$ arm-none-eabi-nm add.elf
... clip ...
00000000 t start
0000000c t stop
```

Note the address assignment for the labels `start` and `stop`. The address assigned for `start` is `0x0`. Since it is the label of the first instruction. The label `stop` is after 3 instructions. Each instructions is 4 bytes. Hence `stop` is assigned an address `12` (`0xC`).

Linking with a different base address for the instructions will result in a different set of addresses being assigned to the labels.

```
$ arm-none-eabi-ld -Ttext=0x20000000 -o add.elf add.o
$ arm-none-eabi-nm add.elf
... clip ...
20000000 t start
2000000c t stop
```

The output file created by `ld` is in a format called `ELF`. Various file formats are available for storing executable code. The ELF format works fine when you have an OS around, but since we are going to run the program on bare metal, we will have to convert it to a simpler file format called the `binary` format.

A file in `binary` format contains consecutive bytes from a specific memory address. No other additional information is stored in the file. This is convenient for Flash programming tools, since all that has to be done when programming is to copy each byte in the file, to consecutive address starting from a specified base address in memory.

The GNU toolchain's `objcopy` command can be used to convert between different object file formats. A common usage of the command is given below.

```
objcopy -O <output-format> <in-file> <out-file>
```

To convert `add.elf` to binary format the following command can be used.

```
$ arm-none-eabi-objcopy -O binary add.elf add.bin
```

Check the size of the file. The file will be exactly 16 bytes. Since there are 4 instructions and each instruction occupies 4 bytes.

```
$ ls -al add.bin
-rw-r--r-- 1 vijaykumar vijaykumar 16 2008-10-03 23:56 add.bin
```

3.2. Executing in Qemu

When the ARM processor is reset, it starts executing from address `0x0`. On the connex board a 16MB Flash is located at address `0x0`. The instructions present in the beginning of the Flash will be executed.

When `qemu` emulates the connex board, a file has to be specified which will be treated file as Flash memory. The Flash file format is very simple. To get the byte from address X in the Flash, `qemu` reads the byte from offset X in the file. In fact, this is the same as the binary file format.

To test the program, on the emulated Gumstix connex board, we first create a 16MB file representing the Flash. We use the `dd` command to copy 16MB of zeroes from `/dev/zero` to the file `flash.bin`. The data is copied in 4K blocks.

```
$ dd if=/dev/zero of=flash.bin bs=4096 count=4096
```

`add.bin` file is then copied into the beginning of the Flash, using the following command.

```
$ dd if=add.bin of=flash.bin bs=4096 conv=notrunc
```

This is the equivalent of programming the `bin` file on to the Flash memory.

After reset, the processor will start executing from address `0x0`, and the instructions from the program will get executed. The command to invoke `qemu` is given below.

```
$ qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/nu
```

The `-M connex` option specifies that the machine `connex` is to be emulated. The `-pflash` options specifies that `flash.bin` file represents the Flash memory. The `-nographic` specifies that simulation of a graphical display is not required. The `-serial /dev/null` specifies that the serial port of the connex board is to be connected to `/dev/null`, so that the serial port data is discarded.

The system executes the instructions and after completion, keeps looping infinitely in the `stop: b stop` instruction. To view the contents of the registers, the monitor interface of `qemu` can be used. The monitor interface is a command line interface, through which the emulated system can be controlled and the status of the system can be viewed. When `qemu` is started with the above mentioned command, the monitor interface is provided in the standard I/O of `qemu`.

To view the contents of the registers the `info registers` monitor command can be used.

```
(qemu) info registers
R00=00000005 R01=00000004 R02=00000009 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=00000000 R14=00000000 R15=0000000c
PSR=400001d3 -Z-- A svc32
```

Note the value in register `R02`. The register contains the result of the addition and should match with the expected value of 9.

3.3. More Monitor Commands

Some useful `qemu` monitor commands are listed in the following table.

Command	Purpose
`help`	List available commands
`quit`	Quits the emulator
`xp /fmt addr`	Physical memory dump from `addr`
`system_reset`	Reset the system.

The `xp` command deserves more explanation. The `fmt` argument specifies how the memory contents is to be displayed. The syntax of `fmt` is `<count><format><size>`.

`count`
: specifies no. of data items to be dumped.

`size`
: specifies the size of each data item. `b` for 8 bits, `h` for 16 bits, `w` for 32 bits and `g` for 64 bits.

`format`
: specifies the display format. `x` for hex, `d` for signed decimal, `u` for unsigned decimal, `o` for octal, `c` for char and `i` for asm instructions.

This `xp` command with the `i` format, can be used to disassemble the instructions present in memory. To disassemble the instructions located at `0x0`, the `xp` command with the `fmt` specified as `4iw` can be used. The `4` specifies 4 items are to be displayed, `i` specifies that the items are to be printed as instructions (yes, a built in disassembler!), `w` specifies that the items are 32 bits in size. The output of the command is shown below.

```
(qemu) xp /4iw 0x0
0x00000000:  mov        r0, #5  ; 0x5
0x00000004:  mov        r1, #4  ; 0x4
0x00000008:  add        r2, r1, r0
0x0000000c:  b  0xc
```

2. Setting up the ARM Lab

4. More Assembler Directives


4. More Assembler Directives

In this section, we will describe some commonly used assembler directives, using two example programs.

1. A program to sum an array
2. A program to calculate the length of a string

4.1. Sum an Array

The following code sums an array of bytes and stores the result in `r3`.

Listing 2. Sum an Array

```
        .text
entry:  b start                 @ Skip over the data
arr:    .byte 10, 20, 25        @ Read-only array of bytes
eoa:                            @ Address of end of array + 1

        .align
start:
        ldr   r0, =eoa          @ r0 = &eoa
        ldr   r1, =arr          @ r1 = &arr
        mov   r3, #0            @ r3 = 0
loop:   ldrb  r2, [r1], #1      @ r2 = *r1++
        add   r3, r2, r3        @ r3 += r2
        cmp   r1, r0            @ if (r1 != r2)
        bne   loop              @    goto loop
stop:   b stop
```

The code introduces two new assembler directives — `.byte` and `.align`. These assembler directives are described below.

4.1.1. `.byte` Directive

The byte sized arguments of `.byte` are assembled into consecutive bytes in memory. There are similar directives `.2byte` and `.4byte` for storing 16 bit values and 32 bit values, respectively. The general syntax is given below.

```
.byte   exp1, exp2, ...
.2byte  exp1, exp2, ...
.4byte  exp1, exp2, ...
```

The arguments could be simple integer literal, represented as binary (prefixed by `0b` or `0B`), octal (prefixed by `0`), decimal or hexadecimal (prefixed by `0x` or `0X`). The integers could also be represented as character constants (character surrounded by single quotes), in which case the ASCII value of the character will be used.

The arguments could also be C expressions constructed out of literals and other symbols. Examples are shown below.

```
pattern:  .byte 0b01010101, 0b00110011, 0b00001111
npattern: .byte npattern - pattern
halpha:   .byte 'A', 'B', 'C', 'D', 'E', 'F'
dummy:    .4byte 0xDEADBEEF
nalpha:   .byte 'Z' - 'A' + 1
```

4.1.2. `.align` Directive

ARM requires that the instructions be present in 32-bit aligned memory locations. The address of the first byte, of the 4 bytes in an instruction, should be a multiple of 4. To adhere to this, the `.align` directive can be used to insert padding bytes till the next byte address will be a multiple of 4. This is required only when data bytes or half words are inserted within code.

4.2. String Length

The following code calculates the length of string and stores the length in register `r1`.

Listing 3. String Length

```
        .text
        b start

str:    .asciz "Hello World"

        .equ   nul, 0

        .align
start:  ldr   r0, =str          @ r0 = &str
        mov   r1, #0

loop:   ldrb  r2, [r0], #1      @ r2 = *(r0++)
        add   r1, r1, #1        @ r1 += 1
        cmp   r2, #nul          @ if (r1 != nul)
        bne   loop              @    goto loop

        sub   r1, r1, #1        @ r1 -= 1
stop:   b stop
```

The code introduces two new assembler directives - `.asciz` and `.equ`. The assembler directives are described below.

4.2.1. `.asciz` Directive

The `.asciz` directive accepts string literals as arguments. String literal are a sequence characters in double quotes. The string literals are assembled into consecutive memory locations. The assembler automatically inserts a `nul` character (*\0* character) after each string.

The `.ascii` directive is same as `.asciz`, but the assembler does not insert a `nul` character after each string.

4.2.2. `.equ` Directive

The assembler maintains something called a symbol table. The symbol table maps label names to addresses. Whenever the assembler encounters a label definition, the assembler makes an entry in the symbol table. And whenever the

assembler encounters a label reference, it replaces the label by the corresponding address from the symbol table.

Using the assembler directive `.equ`, it is also possible to manually insert entries in the symbol table, to map names to values, which are not necessarily addresses. Whenever the assembler encounters these names, it replaces them by their corresponding values. These names and label names are together called symbol names.

The general syntax of the directive is given below.

```
.equ name, expression
```

The `name` is a symbol name, and has the same restrictions as that of the label name. The `expression` could be simple literal, or an expression as explained for the `.byte` directive.

Note

Unlike the `.byte` directive, the `.equ` directive itself does not allocate any memory. They just create entries in the symbol table.


3. Hello ARM

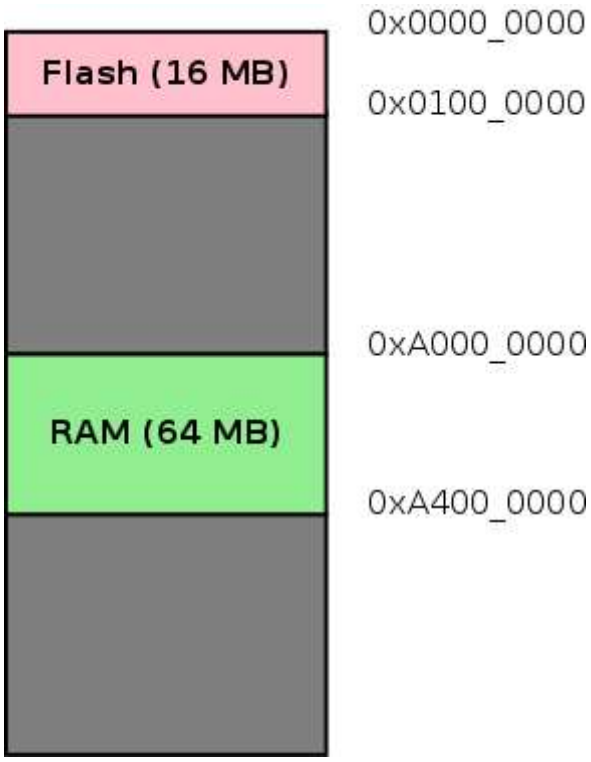

5. Using RAM


5. Using RAM

The Flash memory, in which the previous example programs were stored, is a kind of EEPROM. It is a useful secondary storage, like a hard disk, but is not convenient to store variables in Flash. The variables should be stored in RAM, so that they can be easily modified.

The connex board has a 64 MB of RAM starting at address `0xA0000000`, in which variables can be stored. The memory map of the connex board can be pictured as shown in the following diagram.

Figure 1. Memory Map



Necessary setup has to be done to place the variables at this address. To understand what has to be done, the role of assembler and linker has to be understood.

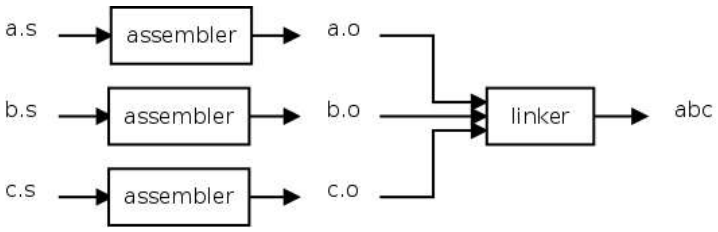

4. More Assembler Directives

6. Linker


6. Linker

While writing a multi-file program, each file is assembled individually into object files. The linker combines these object files to form the final executable.

Figure 2. Role of the Linker



While combining the object files together, the linker performs the following operations.

1. Symbol Resolution
2. Relocation

We will look into these operations, in detail, in this section.

6.1. Symbol Resolution

In a single file program, while producing the object file, all references to labels are replaced by their corresponding addresses by the assembler. But in a multi-file program, if there are any references to labels defined in another file, the assembler marks these references as "unresolved". When these object files are passed to the linker, the linker determines the values for these references from the other object files, and patches the code with the correct values.

The sum of array example is split into two files, to demonstrate the symbol resolution performed by the linker. The two files will be assembled and their symbol tables examined to show the presence of unresolved references.

The file `sum-sub.s` contains the `sum` subroutine, and the file `main.s` invokes the subroutine with the required arguments. The source of the files is shown below.

Listing 4. `main.s` - Subroutine Invocation

```
        .text
        b start                 @ Skip over the data
arr:    .byte 10, 20, 25        @ Read-only array of bytes
eoa:                            @ Address of end of array + 1

        .align
start:
        ldr   r0, =arr          @ r0 = &arr
        ldr   r1, =eoa          @ r1 = &eoa

        bl    sum               @ Invoke the sum subroutine

stop:   b stop
```

Listing 5. `sum-sub.s` - Subroutine Definition

```
        @ Args
        @ r0: Start address of array
        @ r1: End address of array
        @
        @ Result
        @ r3: Sum of Array
```

```
        .global sum

sum:    mov   r3, #0            @ r3 = 0
loop:   ldrb  r2, [r0], #1      @ r2 = *r0++    ; Get array element
        add   r3, r2, r3        @ r3 += r2      ; Calculate sum
        cmp   r0, r1            @ if (r0 != r1) ; Check if hit end-of-ar
        bne   loop              @    goto loop  ; Loop
        mov   pc, lr            @ pc = lr       ; Return when done
```

A word on the `.global` directive is in order. In C, all variables declared outside functions are visible to other files, until explicitly stated as `static`. In assembly, all labels are `static` AKA local (to the file), until explicitly stated that they should be visible to other files, using the `.global` directive.

The files are assembled, and the symbol tables are dumped using the `nm` command.

```
$ arm-none-eabi-as -o main.o main.s
$ arm-none-eabi-as -o sum-sub.o sum-sub.s
$ arm-none-eabi-nm main.o
00000004 t arr
00000007 t eoa
00000008 t start
00000018 t stop
         U sum
$ arm-none-eabi-nm sum-sub.o
00000004 t loop
00000000 T sum
```

For now, focus on the letter in the second column, which specifies the symbol type. A `t` indicates that the symbol is defined, in the text section. A `u` indicates that the symbol is undefined. A letter in uppercase indicates that the symbol is `.global`.

It is evident that the symbol `sum` is defined in `sum-sub.o` and is not resolved yet in `main.o`. When the linker is invoked the symbol references will be resolved, and the executable will be produced.

6.2. Relocation

Relocation is the process of changing addresses already assigned to labels. This will also involve patching up all label references to reflect the newly assigned address. Primarily, relocation is performed for the following two reasons:

1. Section Merging
2. Section Placement

To understand the process of relocation, an understanding of the concept of sections is essential.

Code and data have different run time requirements. For example code can be placed in read-only memory, and data might require read-write memory. It would be convenient, if code and data is **not** interleaved. For this purpose, programs are divided into sections. Most programs have at least two sections, `.text` for code and `.data` for data. Assembler directives `.text` and `.data`, are used to switch back and forth between the two sections.

It helps to imagine each section as a bucket. When the assembler hits a section directive, it puts the code/data following the directive in the selected bucket. Thus the code/data that belong to particular section appear in contiguous locations. The following figures show how the assembler re-arranges data into sections.

Figure 3. Sections

Now that we have an understanding of sections, let us look into the primary reasons for which relocation is performed.

6.2.1. Section Merging

When dealing with multi-file programs, the sections with the same name (example `.text`) might appear, in each file. The linker is responsible for merging sections from the input files, into sections of the output file. By default, the sections, with the same name, from each file is placed contiguously and the label references are patched to reflect the new address.

The effects of section merging can be seen by looking at the symbol table of the object files and the corresponding executable file. The multi-file sum of array program can be used to illustrate section merging. The symbol table of the object files `main.o` and `sum-sub.o` and the symbol table of the executable file `sum.elf` is shown below.

```
$ arm-none-eabi-nm main.o
00000004 t arr
00000007 t eoa
00000008 t start
00000018 t stop
         U sum
$ arm-none-eabi-nm sum-sub.o
00000004 t loop ❶
00000000 T sum
$ arm-none-eabi-ld -Ttext=0x0 -o sum.elf main.o sum-sub.o
$ arm-none-eabi-nm sum.elf
...
00000004 t arr
00000007 t eoa
00000008 t start
00000018 t stop
00000028 t loop ❷
00000024 T sum
```

❶ ❷ The `loop` symbol has address `0x4` in `sum-sub.o`, and `0x28` in `sum.elf`, since the `.text` section of `sum-sub.o` is placed right after the `.text` section of `main.o`.

6.2.2. Section Placement

When a program is assembled, each section is assumed to start from address 0. And thus labels are assigned values relative to start of the section. When the final executable is created, the section is placed at some address X. And all references to the labels defined within the section, are incremented by X, so that they point to the new location.

The placement of each section at a particular location in memory and the patching of all references to the labels in the section, is done by the linker.

The effects of section placement can be seen by looking at the symbol table of the object file and the corresponding executable file. The single file sum of array program can be used to illustrate section placement. To make things clearer, we will place the `.text` section at address `0x100`.

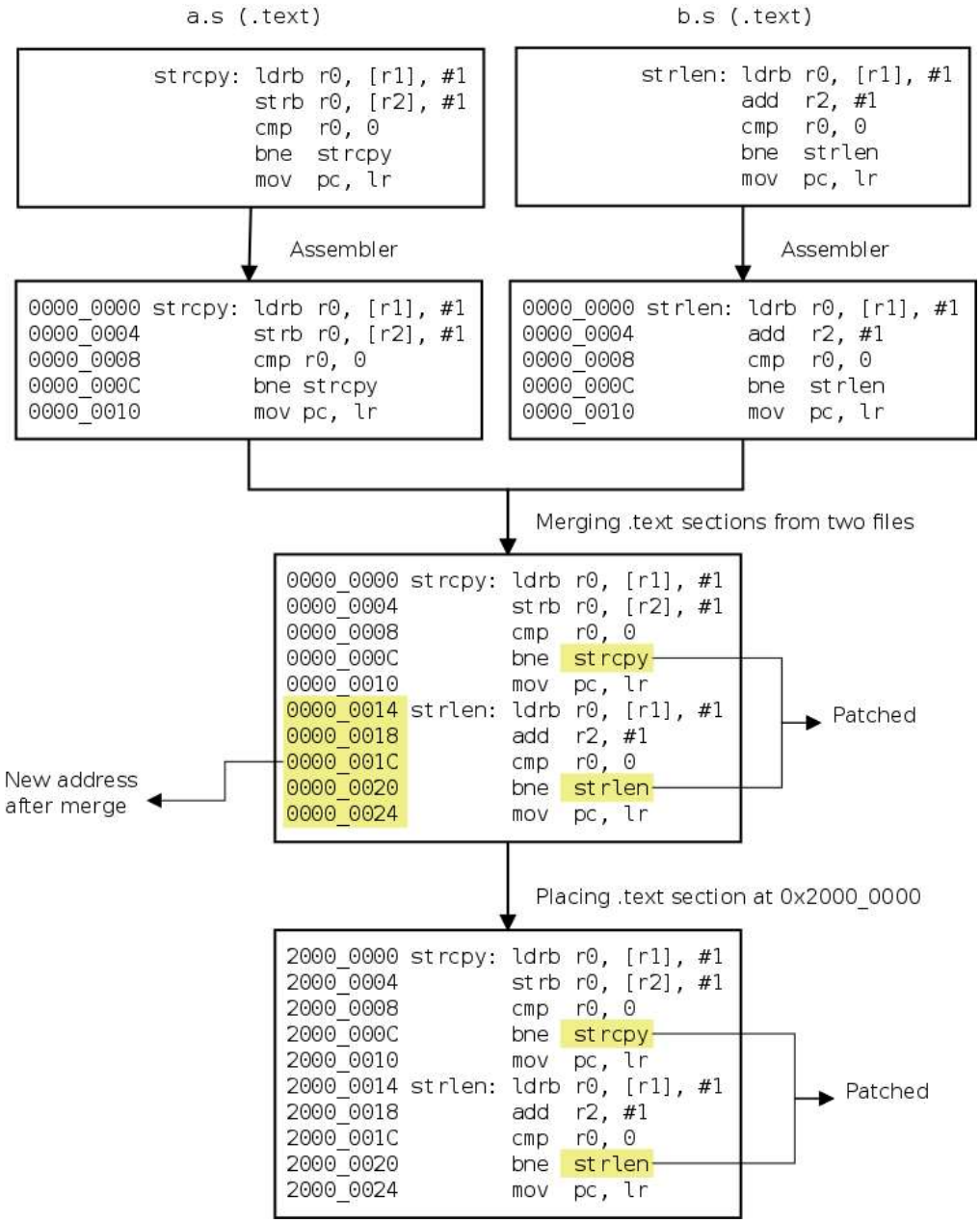
```
$ arm-none-eabi-as -o sum.o sum.s
$ arm-none-eabi-nm -n sum.o
00000000 t entry ❶
00000004 t arr
00000007 t eoa
00000008 t start
00000014 t loop
00000024 t stop
$ arm-none-eabi-ld -Ttext=0x100 -o sum.elf sum.o ❷
$ arm-none-eabi-nm -n sum.elf
00000100 t entry ❸
00000104 t arr
00000107 t eoa
00000108 t start
00000114 t loop
00000124 t stop
...
```

❶ The address for labels are assigned starting from `0` within a section.

2. When the executable is created the linker is instructed to place the text section at address `0x100`.
3. The address for labels in the `.text` section are re-assigned starting from `0x100`, and all label references will be patched to reflect this.

The process of section merging and placement is shown in the following figure.

Figure 4. Section Merging and Placement



5. Using RAM 7. Linker Script File


7. Linker Script File

As mentioned in the previous section, section merging and placement is done by the linker. The programmer can control how the sections are merged, and at what locations they are placed in memory through a linker script file. A very simple linker script file, is shown below.

Listing 6. Basic linker script

```
SECTIONS { ❶
        . = 0x00000000; ❷
        .text : { ❸
                abc.o (.text);
                def.o (.text);
        } ❹
}
```

❶ The `SECTIONS` command is the most important linker command, it specifies how the sections are to be merged and at what location they are to be placed.

❷ Within the block following the `SECTIONS` command, the `.` (period) represents the location counter. The location is always initialised to `0x0`. It can be modified by assigning a new value to it. Setting the value to `0x0` at the beginning is superfluous.

❸ ❹ This part of the script specifies that, the `.text` section from the input files `abc.o` and `def.o` should go to the `.text` section of the output file.

The linker script can be further simplified and generalised by using the wild card character `*` instead of individually specifying the file names.

Listing 7. Wildcard in linker scripts

```
SECTIONS {
        . = 0x00000000;
        .text : { * (.text); }
}
```

If the program contains both `.text` and `.data` sections, the `.data` section merging and location can be specified as shown below.

Listing 8. Multiple sections in linker scripts

```
SECTIONS {
         . = 0x00000000;
         .text : { * (.text); }

         . = 0x00000400;
```

```
        .data : { * (.data); }
}
```

Here, the `.text` section is located at `0x0` and `.data` is located at `0x400`. Note that, if the location counter is not assigned a different value, the `.text` and `.data` sections will be located at adjacent memory locations.

7.1. Linker Script Example

To demonstrate the use of linker scripts, we will use the linker script shown in Listing 8, "Multiple sections in linker scripts" to control the placement of a program's `.text` and `.data` section. We will use a slightly modified version of the sum of array program for this purpose. The code is shown below.

```
        .data
arr:    .byte 10, 20, 25          @ Read-only array of bytes
eoa:                              @ Address of end of array + 1

        .text
start:
        ldr   r0, =eoa            @ r0 = &eoa
        ldr   r1, =arr            @ r1 = &arr
        mov   r3, #0              @ r3 = 0
loop:   ldrb  r2, [r1], #1        @ r2 = *r1++
        add   r3, r2, r3          @ r3 += r2
        cmp   r1, r0              @ if (r1 != r2)
        bne   loop                @    goto loop
stop:   b stop
```

The only change here is that the array is now in the `.data` section. Also note that the nasty branch instruction to skip over the data is also not required, since the linker script will place the `.text` section and `.data` section appropriately. As a result, statements can be placed in the program, in any convenient way, and the linker script will take care of placing the sections correctly in memory.

When the program is linked, the linker script is passed as an input to the linker, as shown in the following command.

```
$ arm-none-eabi-as -o sum-data.o sum-data.s
$ arm-none-eabi-ld -T sum-data.lds -o sum-data.elf sum-data.o
```

The option `-T sum-data.lds` specifies that `sum-data.lds` is to be used as the linker script. Dumping the symbol table, will provide an insight into how the sections are placed in memory.

```
$ arm-none-eabi-nm -n sum-data.elf
00000000 t start
0000000c t loop
0000001c t stop
00000400 d arr
00000403 d eoa
```

From the symbol table it is obvious that the `.text` is placed starting from address `0x0` and `.data` section is placed starting from address `0x400`.

6. Linker

8. Data in RAM, Example


8. Data in RAM, Example

Now that we know, how to write linker scripts, we will attempt to write a program, and place the `.data` section in RAM.

The add program is modified to load two values from RAM, add them and store the result back to RAM. The two values and the space for result is placed in the `.data` section.

Listing 9. Add Data in RAM

```
        .data
val1:   .4byte 10               @ First number
val2:   .4byte 30               @ Second number
result: .4byte 0                @ 4 byte space for result

        .text
        .align
start:
        ldr   r0, =val1         @ r0 = &val1
        ldr   r1, =val2         @ r1 = &val2

        ldr   r2, [r0]          @ r2 = *r0
        ldr   r3, [r1]          @ r3 = *r1

        add   r4, r2, r3        @ r4 = r2 + r3

        ldr   r0, =result       @ r0 = &result
        str   r4, [r0]          @ *r0 = r4

stop:   b stop
```

When the program is linked, the linker script shown below is used.

```
SECTIONS {
        . = 0x00000000;
        .text : { * (.text); }

        . = 0xA0000000;
        .data : { * (.data); }
}
```

The dump of the symbol table of `.elf` is shown below.

```
$ arm-none-eabi-nm -n add-mem.elf
00000000 t start
0000001c t stop
a0000000 d val1
a0000001 d val2
a0000002 d result
```

The linker script seems to have solved the problem of placing the `.data` section in RAM. But wait, the solution is not complete yet!

8.1. RAM is Volatile!

RAM is volatile memory, and hence it is not possible to directly make the data available in RAM, on power up.

All code and data **should** be stored in Flash before power-up. On power-up, a startup code is supposed to copy the data from Flash to RAM, and then proceed with the execution of the program. So the program's `.data` section has two addresses, a **load address** in Flash and a **run-time address** in RAM.

Tip

In `ld` parlance, the load address is called LMA (Load Memory Address), and the run-time address is called VMA (Virtual Memory Address.).

The following two modifications have to be done, to make the program work correctly.

1. The linker script has to be modified to specify both the load address and the run-time address, for the `.data` section.
2. A small piece of code should copy the `.data` section from Flash (load address) to RAM (run-time address).

8.2. Specifying Load Address

The run-time address is what that should be used for determining the address of labels. In the previous linker script, we have specified the run-time address for the `.data` section. The load address is not explicitly specified, and defaults to the run-time address. This is OK, with the previous examples, since the programs were executed directly from Flash. But, if data is to be placed in RAM during execution, the load address should correspond to Flash and the run-time address should correspond to RAM.

A load address different from the run-time address can be specified using the `AT` keyword. The modified linker script is shown below.

```
SECTIONS {
        . = 0x00000000;
        .text : { * (.text); }
        etext = .; ❶

        . = 0xA0000000;
        .data : AT (etext) { * (.data); } ❷
}
```

❶ Symbols can be created on the fly within the `SECTIONS` command by assigning values to them. Here `etext` is assigned the value of the location counter at that position. `etext` contains the address of the next free location in Flash right after all the code. This will be used later on to specify where the `.data` section is to be placed in Flash. Note that `etext` itself will not be allocated any memory, it is just an entry in the symbol table.

❷ The `AT` keyword specifies the load address of the `.data` section. An address or symbol (whose value is a valid address) could be passed as argument to `AT`. Here the load address of `.data` is specified as the location right after all the code in Flash.

8.3. Copying `.data` to RAM

To copy the data from Flash to RAM, the following information is required.

1. Address of data in Flash (`flash_sdata`)
2. Address of data in RAM (`ram_sdata`)
3. Size of the `.data` section. (`data_size`)

With this information the data can be copied from Flash to RAM using the following code snippet.

```
        ldr   r0, =flash_sdata
        ldr   r1, =ram_sdata
        ldr   r2, =data_size

copy:
        ldrb  r4, [r0], #1
        strb  r4, [r1], #1
        subs  r2, r2, #1
        bne   copy
```

The linker script can be slightly modified to provide these information.

Listing 10. Linker Script with Section Copy Symbols

```
SECTIONS {
        . = 0x00000000;
        .text : {
              * (.text);
        }
        flash_sdata = .; ❶

        . = 0xA0000000;
        ram_sdata = .; ❷
        .data : AT (flash_sdata) {
              * (.data);
        };
        ram_edata = .; ❸
        data_size = ram_edata - ram_sdata; ❹
}
```

❶ Start of data in Flash is right after all the code in Flash.

❷ Start of data in RAM is at the base address of RAM.

❸ ❹ Obtaining the size of data is not straight forward. The data size is calculated from the difference in the start of data in RAM and the end of data in RAM. Yes, simple expressions are allowed within the linker script.

The add program with data copied to RAM from Flash is listed below.

Listing 11. Add Data in RAM (with copy)

```
        .data
val1:   .4byte 10               @ First number
val2:   .4byte 30               @ Second number
result: .space 4                @ 1 byte space for result

        .text

        ;; Copy data to RAM.
start:
        ldr   r0, =flash_sdata
        ldr   r1, =ram_sdata
        ldr   r2, =data_size

copy:
        ldrb  r4, [r0], #1
        strb  r4, [r1], #1
        subs  r2, r2, #1
        bne   copy

        ;; Add and store result.
        ldr   r0, =val1         @ r0 = &val1
        ldr   r1, =val2         @ r1 = &val2

        ldr   r2, [r0]          @ r2 = *r0
        ldr   r3, [r1]          @ r3 = *r1

        add   r4, r2, r3        @ r4 = r2 + r3

        ldr   r0, =result       @ r0 = &result
        str   r4, [r0]          @ *r0 = r4

stop:   b stop
```

The program is assembled and linked using the linker script listed in Listing 10, “Linker Script with Section Copy Symbols”. The program is executed and tested within Qemu.

```
qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/nul
(qemu) xp /4dw 0xA0000000
a0000000:          10          30          40           0
```

Note

In a real system with an SDRAM, the memory should not be accessed right-away. The memory controller will have to be initialised before performing a memory access. Our code works because the simulated memory does not require the memory controller to be initialised.

7. Linker Script File

9. Exception Handling


9. Exception Handling

The examples given so far have a major bug. The first 8 words in the memory map are reserved for the exception vectors. When an exception occurs the control is transferred to one these 8 locations. The exceptions and their exception vector addresses are show in the following table.

Table 1. Exception Vector Addresses

Exception	Address
Reset	0x00
Undefined Instruction	0x04
Software Interrupt (SWI)	0x08
Prefetch Abort	0x0C
Data Abort	0x10
Reserved, not used	0x14
IRQ	0x18
FIQ	0x1C

These locations are supposed to contain a branch that will transfer control the appropriate exception handler. In the examples we have seen so far, we haven't inserted branch instructions at the exception vector addresses. We got away without issues since these exceptions did not occur. All the above programs can be fixed, by linking them with the following assembly code.

```
        .section "vectors"
reset:  b       start
undef:  b       undef
swi:    b       swi
pabt:   b       pabt
dabt:   b       dabt
        nop
irq:    b       irq
fiq:    b       fiq
```

Only the reset exception is vectored to a different address `start`. All other exceptions are vectored to the same address. So if any exception other that reset occurs, the processor will be spinning in the same location. The exception can then be identified by looking at the value of `pc` through a debugger (the monitor interface in our case).

To ensure that these instruction are placed at the exception vector addresses, the linker script should look something like below.

```
SECTIONS {
        . = 0x00000000;
        .text : {
                * (vectors);
                * (.text);
                ...
        }
        ...
}
```

Notice how the `vectors` section is placed before all other code, ensuring that the `vectors` is located at address starting from 0x0.

8. Data in RAM, Example

10. C Startup


10. C Startup

It is not possible to directly execute C code, when the processor comes out of reset. Since, unlike assembly language, C programs need some basic pre-requisites to be satisfied. This section will describe the pre-requisites and how to meet the pre-requisites.

We will take the example of C program that calculates the sum of an array as an example. And by the end of this section, we will be able to perform the necessary setup, transfer control to the C code and execute it.

Listing 12. Sum of Array in C

```
static int arr[] = { 1, 10, 4, 5, 6, 7 };
static int sum;
static const int n = sizeof(arr) / sizeof(arr[0]);

int main()
{
        int i;

        for (i = 0; i < n; i++)
                sum += arr[i];
}
```

Before transferring control to C code, the following have to be setup correctly.

1. Stack
2. Global variables
    a. Initialized
    b. Uninitialized
3. Read-only data

10.1. Stack

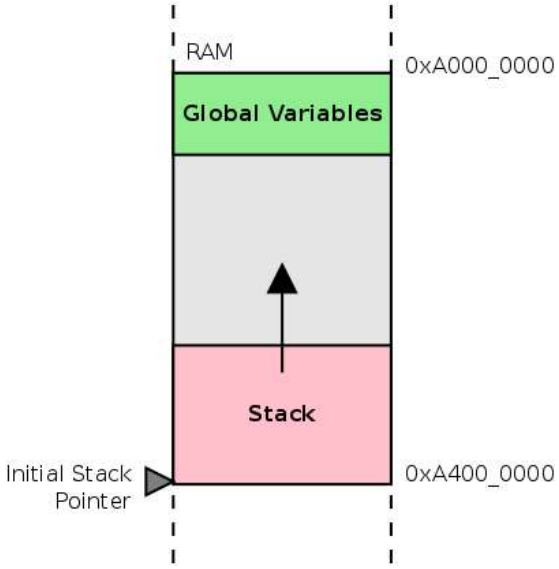
C uses the stack for storing local (auto) variables, passing function arguments, storing return address, etc. So it is essential that the stack be setup correctly, before transferring control to C code.

Stacks are highly flexible in the ARM architecture, since the implementation is completely left to the software. For people not familiar with the ARM architecture a overview is provided in Appendix C, *ARM Stacks*.

To make sure that code generated by different compilers is interroperable, ARM has created the ARM Architecture Procedure Call Standard (AAPCS). The register to be used as the stack pointer and the direction in which the stack grows is all dictated by the AAPCS. According to the AAPCS, **register `r13`** is to be used as the stack pointer. Also the stack should be **full-descending**.

One way of placing global variables and the stack is shown in the following diagram.

Figure 5. Stack Placement



So all that has to be done in the startup code is to point `r13` at the highest RAM address, so that the stack can grow downwards (towards lower addresses). For the `connex` board this can be acheived using the following ARM instruction.

```
        ldr sp, =0xA4000000
```

Note that the the assembler provides an alias `sp` for the `r13` register.

> **Note**
> The address `0xA4000000` itself does not correspond to RAM. The RAM ends at `0xA3FFFFFF`. But that is OK, since the stack is **full**-descending, during the first push the stack pointer will be decremented first and the value will be stored.

10.2. Global Variables

When C code is compiled, the compiler places initialized global variables in the `.data` section. So just as with the assembly, the `.data` has to be copied from Flash to RAM.

The C language guarantees that all uninitialized global variables will be initialized to zero. When C programs are compiled, a separate section called `.bss` is used for uninitialized variables. Since the value of these variables are all zeroes to start with, they do not have to be stored in Flash. Before transferring control to C code, the memory locations corresponding to these variables have to be initialized to zero.

10.3. Read-only Data

GCC places global variables marked as `const` in a separate section, called `.rodata`. The `.rodata` is also used for storing string constants.

Since contents of `.rodata` section will not be modified, they can be placed in Flash. The linker script has to modified to accomodate this.

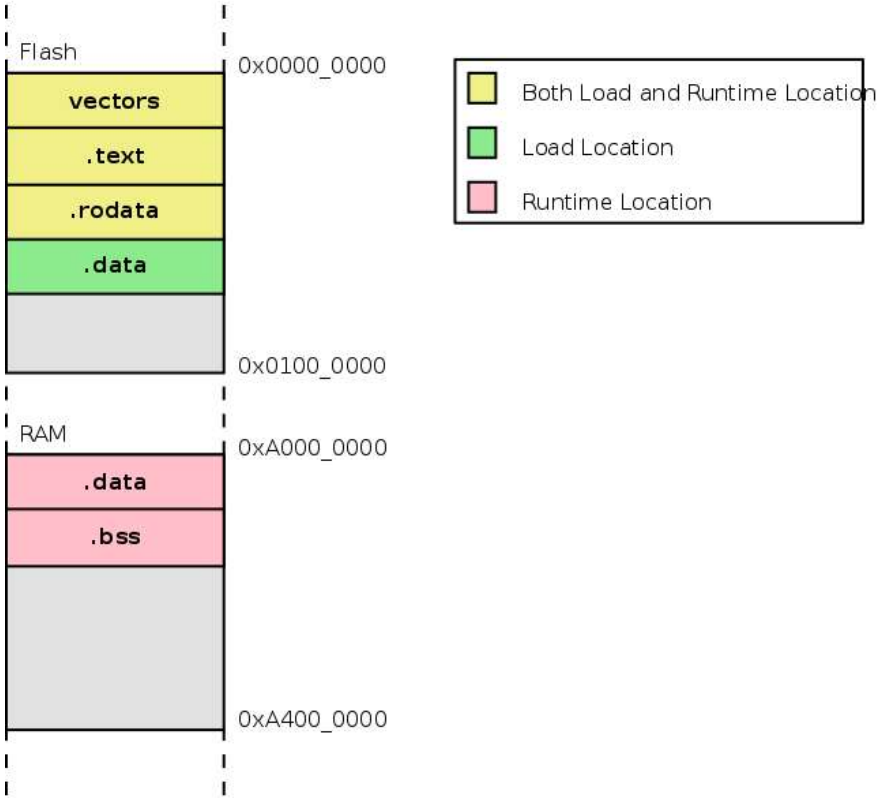
10.4. Startup Code

Now that we know the pre-requisites we can create the linker script and the startup code. The linker script Listing 10, "Linker Script with Section Copy Symbols" is modified to accomodate the following.

1. `.bss` section placement
2. `vectors` section placement
3. `.rodata` section placement

The `.bss` is placed right after `.data` section in RAM. Symbols to locate the start of `.bss` and end of `.bss` are also created in the linker script. The `.rodata` is placed right after `.text` section in Flash. The following diagram shows the placement of the various sections.

Figure 6. Section Placement



Listing 13. Linker Script for C code

```
SECTIONS {
        . = 0x00000000;
        .text : {
              * (vectors);
              * (.text);
        }
        .rodata : {
              * (.rodata);
        }
        flash_sdata = .;

        . = 0xA0000000;
        ram_sdata = .;
        .data : AT (flash_sdata) {
              * (.data);
        }
        ram_edata = .;
        data_size = ram_edata - ram_sdata;

        sbss = .;
        .bss : {
             * (.bss);
        }
        ebss = .;
        bss_size = ebss - sbss;
}
```

The startup code has the following parts

1. exception vectors
2. code to copy the `.data` from Flash to RAM
3. code to zero out the `.bss`
4. code to setup the stack pointer

5. branch to main

Listing 14. C Startup Assembly

```
        .section "vectors"
reset:  b       start
undef:  b       undef
swi:    b       swi
pabt:   b       pabt
dabt:   b       dabt
        nop
irq:    b       irq
fiq:    b       fiq

        .text
start:
        @@ Copy data to RAM.
        ldr   r0, =flash_sdata
        ldr   r1, =ram_sdata
        ldr   r2, =data_size

        @@ Handle data_size == 0
        cmp   r2, #0
        beq   init_bss
copy:
        ldrb   r4, [r0], #1
        strb   r4, [r1], #1
        subs   r2, r2, #1
        bne    copy

init_bss:
        @@ Initialize .bss
        ldr   r0, =sbss
        ldr   r1, =ebss
        ldr   r2, =bss_size

        @@ Handle bss_size == 0
        cmp   r2, #0
        beq   init_stack

        mov   r4, #0
zero:
        strb  r4, [r0], #1
        subs  r2, r2, #1
        bne   zero

init_stack:
        @@ Initialize the stack pointer
        ldr   sp, =0xA4000000

        bl    main

stop:   b       stop
```

To compile the code, it is not necessary to invoke the assembler, compiler and linker individually. `gcc` is intelligent enough to do that for us.

As promised before, we will compile and execute the C code shown in Listing 12, "Sum of Array in C".

```
$ arm-none-eabi-gcc -nostdlib -o csum.elf -T csum.lds csum.c startup.s
```

The `-nostdlib` option is used to specify that the standard C library should not be linked in. A little extra care has to be taken when the C library is linked in. This is discussed in Section 11, "Using the C Library".

A dump of the symbol table will give a better picture of how things have been placed in memory.

```
$ arm-none-eabi-nm -n csum.elf
00000000 t reset          ❶
00000004 A bss_size
00000004 t undef
00000008 t swi
0000000c t pabt
00000010 t dabt
00000018 A data_size
00000018 t irq
0000001c t fiq
00000020 T main
00000090 t start          ❷
000000a0 t copy
000000b0 t init_bss
000000c4 t zero
000000d0 t init_stack
000000d8 t stop
000000f4 r n              ❸
000000f8 A flash_sdata
a0000000 d arr            ❹
a0000000 A ram_sdata
a0000018 A ram_edata
a0000018 A sbss
a0000018 b sum            ❺
a000001c A ebss
```

❶ `reset` and the rest of the exception vectors are placed starting from `0x0`.
❷ The assembly code is placed right after the 8 exception vectors (`8 * 4 = 32 = 0x20`).
❸ The read-only data `n`, is placed in Flash after the code.
❹ The initialized data `arr`, an array of 6 integers, is placed at the start of RAM `0xA0000000`.
❺ The uninitialized data `sum` is placed after the array of 6 integers. (`6 * 4 = 24 = 0x18`)

To execute the program, convert the program to `.bin` format, execute in Qemu, and dump the `sum` variable located at `0xA0000018`.

```
$ arm-none-eabi-objcopy -O binary csum.elf csum.bin
$ dd if=csum.bin of=flash.bin bs=4096 conv=notrunc
$ qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/nu
(qemu) xp /6dw 0xa0000000
a0000000:           1          10           4           5
a0000010:           6           7
(qemu) xp /1dw 0xa0000018
a0000018:          33
```