From Zero to main(): Bare metal C

Working on embedded software, one quickly develops a quasi-religious respect for the axioms of embedded C programming:

1. The entry point of thy program shall be named "main".
2. Thou shalt initialize thy static variables, else The Machine shall set them to zero.
3. Thou shalt implement The Interrupts. HardFault_Handler chief among them, but also SysTick_Handler.

Ask an engineer where those rules come from, and they'll wave towards cryptic startup files implemented in assembly. Often times those files are copy-pasted from project to project. Seldom are they ever read, let alone modified.

Throughout the Zero to main() series of posts, we demystify what happens between when power is applied and your main function is called. In the process, we'll learn how to bootstrap a C environment, implement a bootloader, relocate code, and more!

Setting the stage

While most of the concepts and code presented in this series should work for all Cortex-M series MCUs, our examples target the SAMD21G18 processor by Atmel. This is a Cortex-M0+ chip found on several affordable development boards.

Specifically, we are using:

In each case, we'll implement a simple blinking LED application. It is not particularly interesting in itself, but for the sake of completeness you can find the code reproduced below.

```
#include <samd21g18a.h>

#include <port.h>
#include <stdbool.h>
#include <stdint.h>

#define LED_0_PIN PIN_PA17

static void set_output(const uint8_t pin) {
  struct port_config config_port_pin;
```

```
  port_get_config_defaults(&config_port_pin);
  config_port_pin.direction = PORT_PIN_DIR_OUTPUT;
  port_pin_set_config(pin, &config_port_pin);
  port_pin_set_output_level(pin, false);
}

int main() {
  set_output(LED_0_PIN);
  while (true) {
    port_pin_toggle_output_level(LED_0_PIN);
    for (volatile int i = 0; i < 100000; ++i) {}
  }
}
```

Power on!

So how did we get to main? All we can tell from observation is that we applied power to the board and our code started executing. There must be behavior intrinsic to the chip that defines how code is executed.

And indeed, there is! Digging into the ARMv6-M Technical Reference Manual, which is the underlying architecture manual for the Cortex-M0+, we can find some pseudo-code that describes reset behavior:

```
// B1.5.5 TakeReset()
// ============
TakeReset()
    VTOR = Zeros(32);
    for i = 0 to 12
        R[i] = bits(32) UNKNOWN;
    bits(32) vectortable = VTOR;
    CurrentMode = Mode_Thread;
    LR = bits(32) UNKNOWN; // Value must be initialised by software
    APSR = bits(32) UNKNOWN; // Flags UNPREDICTABLE from reset
    IPSR<5:0> = Zeros(6); // Exception number cleared at reset
    PRIMASK.PM = '0'; // Priority mask cleared at reset
    CONTROL.SPSEL = '0'; // Current stack is Main
    CONTROL.nPRIV = '0'; // Thread is privileged
    ResetSCSRegs(); // Catch-all function for System Control Space reset
    for i = 0 to 511 // All exceptions Inactive
        ExceptionActive[i] = '0';
    ClearEventRegister(); // See WFE instruction for more information
    SP_main = MemA[vectortable,4] AND 0xFFFFFFFC<31:0>;
    SP_process = ((bits(30) UNKNOWN):'00');
```

```
start = MemA[vectortable+4,4]; // Load address of reset routine
BLXWritePC(start); // Start execution of reset routine
```

In short, the chip does the following:

- Reset the vector table address to `0x00000000`
- Disable all interrupts
- Load the SP from address `0x00000000`
- Load the PC from address `0x00000004`

"Mystery solved!", you'll say. Our `main` function must be at address `0x00000004`!

Let us check.

First, we dump our `bin` file to see what address `0x0000000` and `0x00000004` contain:

```
francois-mba:zero-to-main francois$ xxd build/minimal/minimal.bin  | head
00000000: 0020 0020 c100 0000 b500 0000 bb00 0000  . . ............
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

If I'm reading this correctly, our inital SP is `0x20002000`, and our start address pointer is `0x000000c1`.

Let's dump our symbols to see which one is at `0x000000c1`.

```
francois-mba:minimal francois$ arm-none-eabi-objdump -t build/minimal.elf | sort
...
000000b4 g     F .text  00000006 NMI_Handler
000000ba g     F .text  00000006 HardFault_Handler
000000c0 g     F .text  00000088 Reset_Handler
00000148 l     F .text  0000005c system_pinmux_get_group_from_gpio_pin
000001a4 l     F .text  00000020 port_get_group_from_gpio_pin
000001c4 l     F .text  00000022 port_get_config_defaults
000001e6 l     F .text  0000004e port_pin_set_output_level
00000234 l     F .text  00000038 port_pin_toggle_output_level
0000026c l     F .text  00000040 set_output
000002ac g     F .text  0000002c main
...
```

That's odd! Our main function is found at `0x000002ac`. No symbol at `0x000000c1`, but a `Reset_Handler` symbol at `0x000000c0`.

It turns out that the lowest bit of the PC is used to indicate thumb2 instructions, which is one of the two instruction sets supported by ARM processors, so `Reset_Handler` is what we're looking for (for more details check out section A4.1.1 in the ARMv6-M manual).

Writing a Reset_Handler

Unfortunately, the Reset_Handler is often an inscrutable mess of Assembly code. See the nRF52 SDK startup file for example.

Instead of going through this file line-by-line, let's see if we can write a minimal Reset_Handler from first principles.

Here again, ARM's Technical Reference Manuals are useful. Section 5.9.2 of the Cortex-M3 TRM contains the following table:

Reset boot-up behavior

Action	Description
Initialize variables	Any global/static variables must be setup. This includes initializing the BSS variable to 0, and copying initial values from ROM to RAM for non-constant variables.
[Setup stacks]	If more than one stack is be used, the other banked SPs must be initialized. The current SP can also be changed to Process from Main.
[Initialize any runtime]	Optionally make calls to C/C++ runtime init code to enable use of heap, floating point, or other features. This is normally done by `__main` from the C/C++ library.

So, our ResetHandler is responsible for initializing static and global variables, and starting our program. This mirrors what the C standards tells us:

> All objects with static storage duration shall be initialized (set to their initial values) before program startup. The manner and timing of such initialization are otherwise unspecified.

(Section 5.1.2, Execution environment)

In practice this means that given the following snippet:

```
static uint32_t foo;
static uint32_t bar = 2;
```

Our `Reset_Handler` needs to make sure that the memory at `&foo` is 0x00000000, and the memory at `&bar` is 0x00000002.

We cannot just go and initialize each variable one by one. Instead, we rely on the compiler (technically, the linker) to put all those variables in the same place so we can initialize them in one fell swoop.

For static variables that must be zeroed, the linker gives us `_sbss` and `_ebss` as start and end addresses. We can therefore do:

```
/* Clear the zero segment */
for (uint32_t *bss_ptr = &_sbss; bss_ptr < &_ebss;) {
    *bss_ptr++ = 0;
}
```

For static variables with an init value, the linker gives us:

- `_etext` as the address the init values are stored at
- `_sdata` as the address the static variables live at
- `_edata` as the end of the static variables memory

We then can do:

```
uint32_t *init_values_ptr = &_etext;
uint32_t *data_ptr = &_sdata;
if (init_values_ptr != data_ptr) {
    for (; data_ptr < &_edata;) {
        *data_ptr++ = *init_values_ptr++;
    }
}
```

Putting it together, we can write our `Reset_Handler`

```
void Reset_Handler(void)
{
    /* Copy init values from text to data */
    uint32_t *init_values_ptr = &_etext;
    uint32_t *data_ptr = &_sdata;

    if (init_values_ptr != data_ptr) {
        for (; data_ptr < &_edata;) {
            *data_ptr++ = *init_values_ptr++;
        }
    }

    /* Clear the zero segment */
    for (uint32_t *bss_ptr = &_sbss; bss_ptr < &_ebss;) {
        *bss_ptr++ = 0;
    }
}
```

We still need to start our program! That's achieved with a simple call to `main()`.

```
void Reset_Handler(void)
{
    /* Copy init values from text to data */
    uint32_t *init_values_ptr = &_etext;
    uint32_t *data_ptr = &_sdata;

    if (init_values_ptr != data_ptr) {
        for (; data_ptr < &_edata;) {
            *data_ptr++ = *init_values_ptr++;
        }
    }

    /* Clear the zero segment */
    for (uint32_t *bss_ptr = &_sbss; bss_ptr < &_ebss;) {
        *bss_ptr++ = 0;
    }

    /* Overwriting the default value of the NVMCTRL.CTRLB.MANW bit (errata reference 131
    NVMCTRL->CTRLB.bit.MANW = 1;

    /* Branch to main function */
    main();

    /* Infinite loop */
    while (1);
}
```

You will note that we added two things:

1. An infinite loop after `main()`, so we do not run off into the weeds if the main function returns
2. Workaround for chip bugs which are best taken care of before our program starts. Sometimes these are wrapped in a `SystemInit` function called by the `Reset_Handler` before `main`. This is the approach taken by Nordic.

Closing

All the code used in this blog post is available on Github.

See anything you'd like to change? Submit a pull request or open an issue at GitHub

More complex programs often require a more complicated `Reset_Handler`. For example:

1. Relocatable code must be copied over
2. If our program relies on libc, we must initialize it
   EDIT: Post written! - From Zero to main(): Bootstrapping libc with Newlib
3. More complex memory layouts can add a few copy / zero loops

We'll cover all of them in future posts. But before that, we'll talk about how the magical memory region variables come about, how our `Reset_Handler`'s address ends up at `0x00000004`, and how to write a linker script in our next post!

EDIT: Post written! - From Zero to main(): Demystifying Firmware Linker Scripts