FIRST EDITION – Chapter 2

Kevin Thomas

Forward

I remember when I started learning programming to which my first language was 6502 Assembler. It was to program a Commodore 64 and right from the beginning I learned the lowest level development possible.

Literally every piece of the Commodore 64 was understood as it was a simple machine. There was absolutely no abstraction layer of any kind.

Everything we did we had an absolute mastery of however it was a very simple architecture.

Microcontrollers are small systems without an operating system and are also very simple in their design. They are literally everywhere from your toaster to your fridge to your TV and billions of other electronics that you never think about.

Most microcontrollers are developed in the C programming language with has its roots to the 1970’s however dominates the landscape.

Perhaps if Rust ever gains ground in microcontrollers there will be a supplemental book however today in 2023 it is very much a C landscape.

We will take our time and learn the basics of C within an STM32F401CC6 microcontroller.

Below are items you will need for this book.

STM32F401CCU6
https://www.amazon.com/SongHe-STM32F401-Development-STM32F401CCU6-%20Learning/dp/B07XBWGF9M

ST-Link V2 Emulator Downloader Programmer
https://www.amazon.com/HiLetgo-Emulator-Downloader-ProgrammerSTM32F103C8T6/dp/B07SQV6VLZ

DSD TECH HM-11 Bluetooth 4.0 BLE Module
https://www.amazon.com/DSD-TECH-Bluetooth-Compatible-Devices/dp/B07CHNJ1QN

Electronics Soldering Iron Kit
https://www.amazon.com/Electronics-Adjustable-Temperature-ControlledThermostatic/dp/B0B28JQ95M?th=1

Premium Breadboard Jumper Wires
https://www.amazon.com/Keszoox-Premium-Breadboard-Jumper-Raspberry/%20dp/B09F6X3N79

Breadboard Kit
https://www.amazon.com/Breadboards-Solderless-BreadboardDistribution-Connecting/dp/B07DL13RZH

SSD1306 Display
https://www.amazon.com/Hosyond-Display-Self-Luminous-Compatible-Raspberry/dp/B09T6SJBV5

8x8 WS2182 NeoPixel Array
https://www.amazon.com/Tiabiaya-LED-Panel-CJMCU-8X8-Module/dp/B0BLGN88NH

6x6x5mm Momentary Tactile Tact Push Button Switches
https://www.amazon.com/DAOKI-Miniature-Momentary-Tactile-Quality/dp/B01CGMP9GY

NOTE: The item links may NOT be available but the descriptions allow you to shop on any online or physical store of your choosing.

Let’s begin...

Table Of Contents

Chapter 1: hello, world

We begin our journey building the traditional *hello, world* example in Embedded C.

We will then reverse engineer the binary in GDB.

We are going to use PlatformIO and VSCode.

Let’s download the GNU Arm Embedded Toolchain for Windows, MAC or Linux.

https://developer.arm.com/downloads/-/gnu-rm

Let’s download OpenOCD.

https://gnutoolchains.com/arm-eabi/openocd

Let’s download Visual Studio Code which we will use as our integrated development environment.

https://code.visualstudio.com/

Once installed, let’s add the Platform IO IDE extension within VS Code.

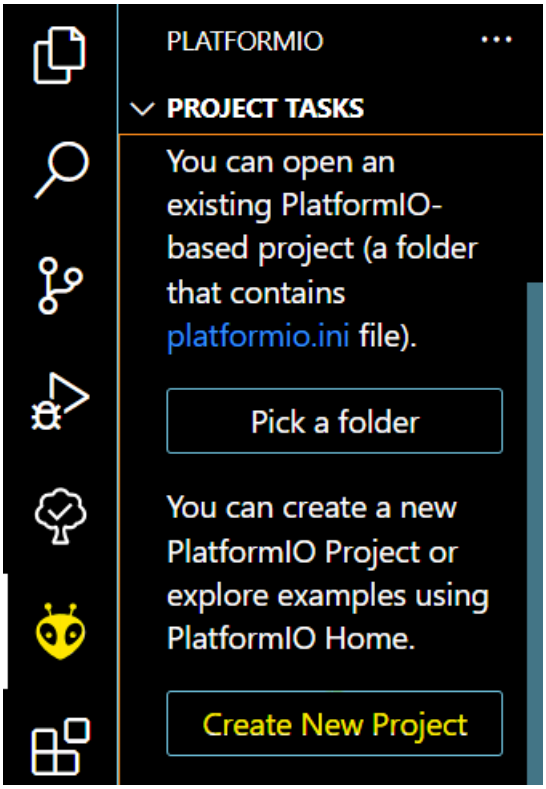
https://marketplace.visualstudio.com/items?itemName=platformio.platformio-ide

Let’s create a new project and get started by following the below steps.

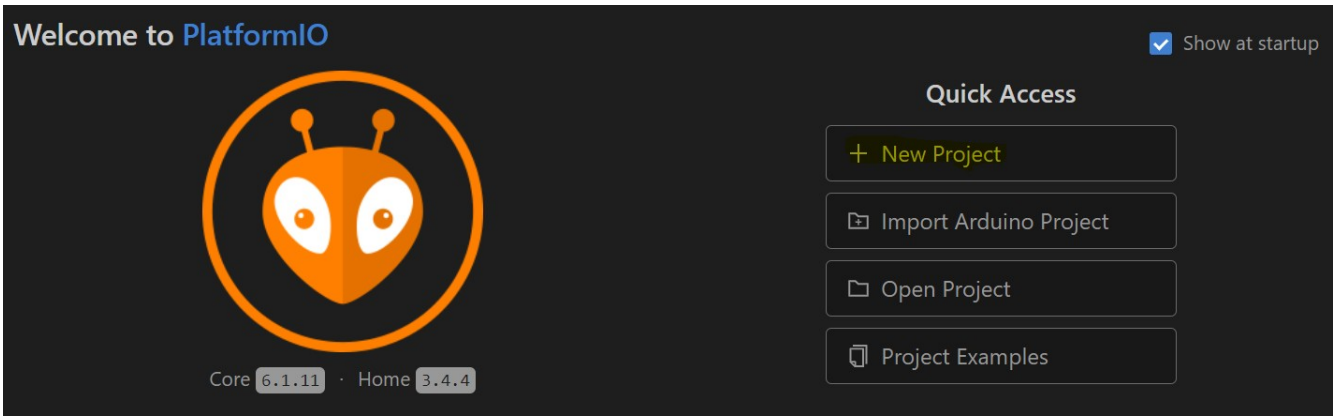
You will see the PlatformIO icon on the left side-bar click on it and it will pop up a menu as below. You will see the items in yellow to help identify what to click on through the process.

Let’s begin!

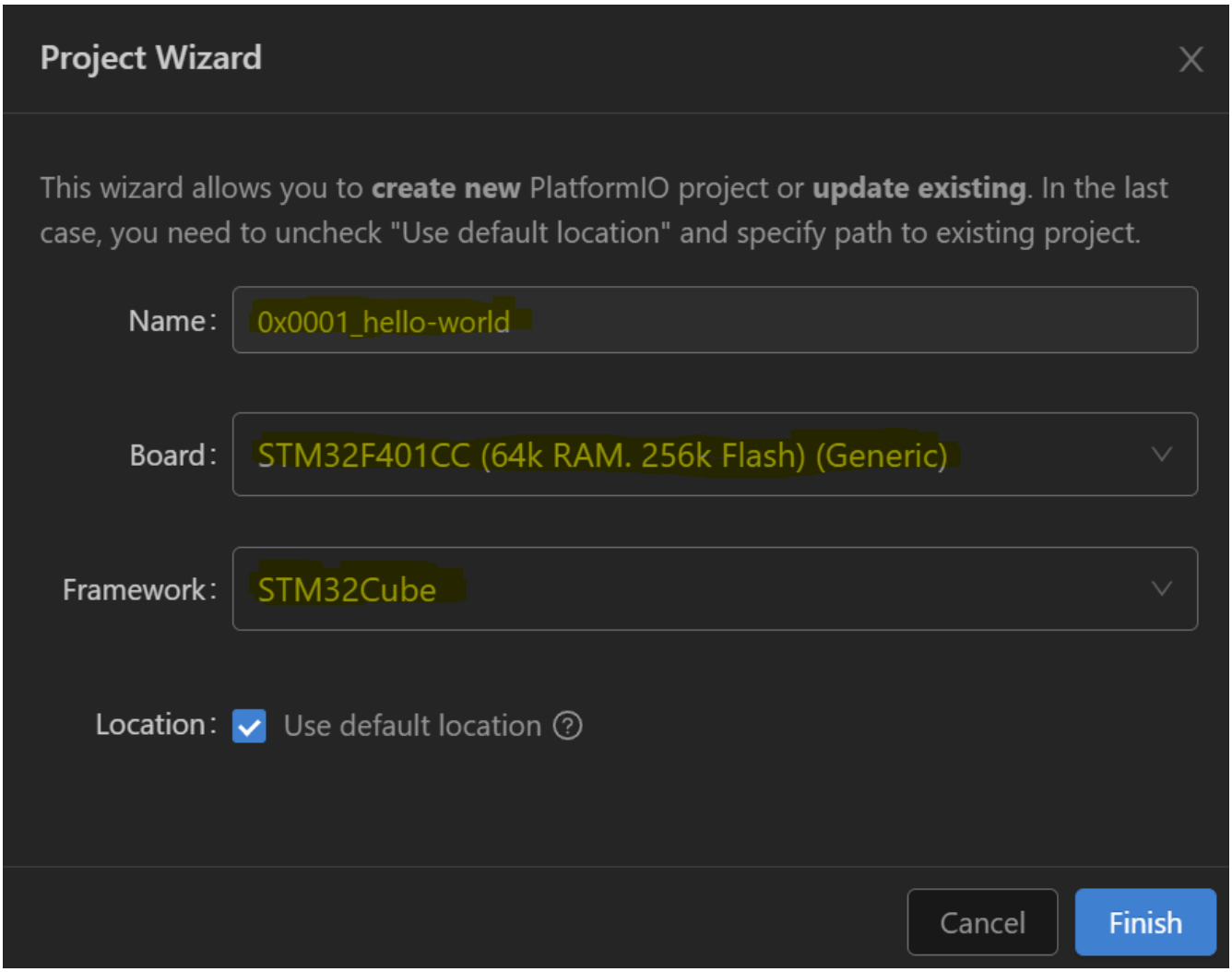
click **Create New Project**



click **New Project**



Name: **0x0001_hello-world**
Board: STM32F401CC
Framework: STM32Cube
click Finish



These are the steps to create a new project however I have created such a folder in the GitHub repo here which has all of the custom libraries needed to work with our example.

If you do not have Git installed, here is a link to install git on Windows, MAC and Linux.

https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Clone the repo to whatever folder you prefer.

git clone https://github.com/mytechnotalent/Embedded-Hacking.git

Open VS Code and click **File** then **Open Folder …** then click on the **Embedded-Hacking** folder and then select **0x0001_hello-world**.

Give it a few minutes to initialize.

If there are any issues I would point you to the below document as it is related to Windows. We are not using the Arduino framework however look for the heading, **Important steps for Windows users, before installing**, to handle the long paths issues in Git for Windows.

https://arduino-pico.readthedocs.io/en/latest/platformio.html

If you have any additional issues here is the docs for the STM32 Platform IO port.

https://docs.platformio.org/en/stable/platforms/ststm32.html#tutorials

Let’s first examine the toolbar on the bottom of VS Code.

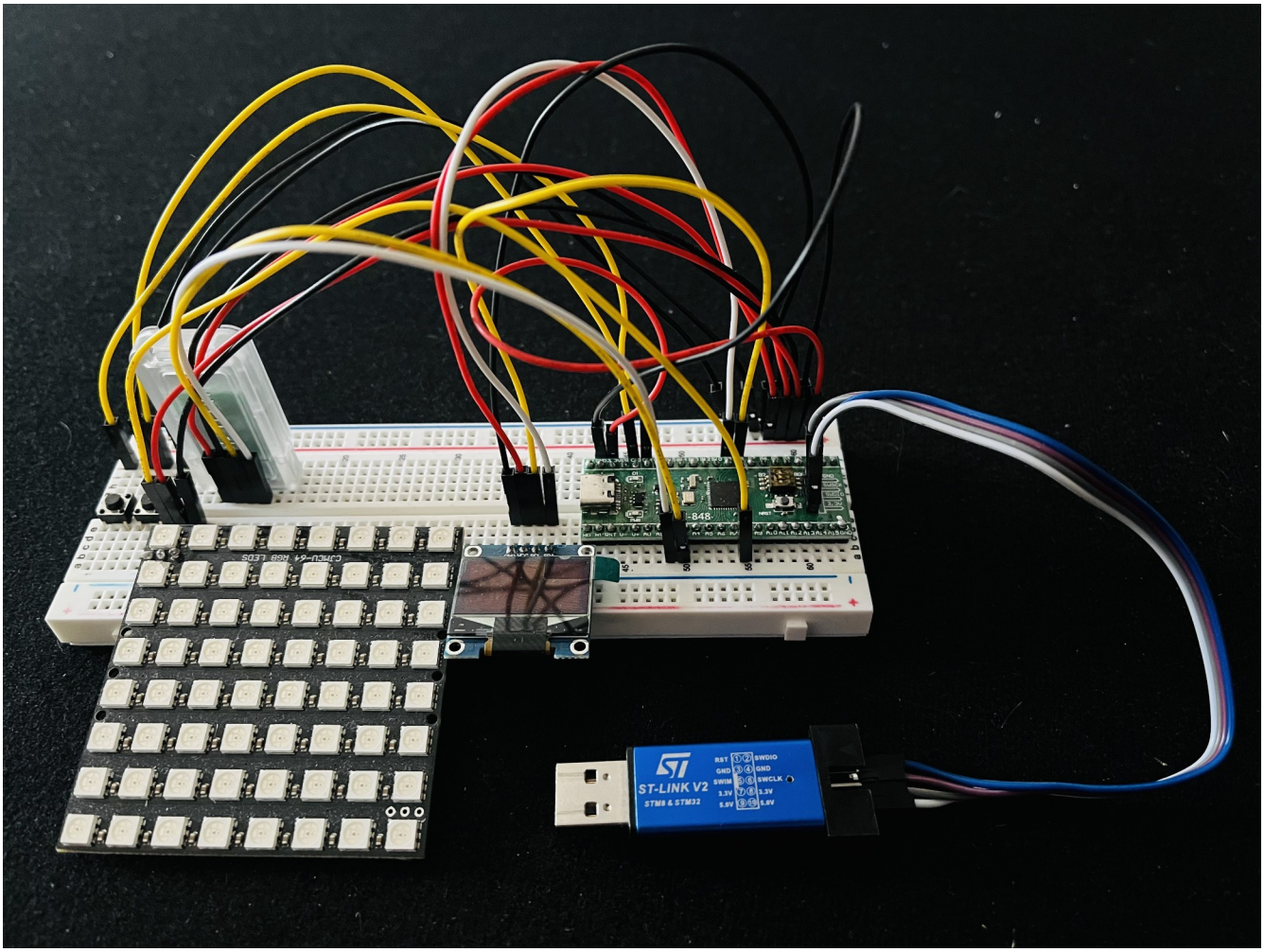
We can compile and upload this code in one button click by clicking on the → button above.

If everything is successful you will see SUCCESS in green within the terminal.

```
[SUCCESS]
```

Before we go any further we need to first soldier up our STM32F401CCU and attach it into the breadboard. We then need to wire up our various external devices.

Let’s start with an overall picture of our setup and I will walk through every connection one-by-one.



NOTE: The item links may NOT be available but the descriptions allow you to shop on any online or physical store of your choosing.

STM32F401CCU6
https://www.amazon.com/SongHe-STM32F401-Development-STM32F401CCU6-%20Learning/dp/B07XBWGF9M

ST-Link V2 Emulator Downloader Programmer
https://www.amazon.com/HiLetgo-Emulator-Downloader-ProgrammerSTM32F103C8T6/dp/B07SQV6VLZ

After soldering up the STM32F401CCU6, place it into the breadboard with the orientation showing above where the USB-C is to the left and not near the right edge. The debug pins are the 4 pins on the right and side of the board.

```
connect the MCU 3.3V DEBUG PIN to the ST-LINK 3.3V PIN
connect the MCU GND DEBUG PIN to the ST-LINK GND PIN
connect the MCU SWDIO DEBUG PIN to the ST-LINK SWDIO PIN
connect the MCU SWSCK DEBUG PIN to the ST-LINk SWSCK PIN
connect the MCU 3.3V to the 3.3V rail of the breadboard PIN
connect the MCU GND to the GND rail of the breadboard PIN
```

DSD TECH HM-11 Bluetooth 4.0 BLE Module
https://www.amazon.com/DSD-TECH-Bluetooth-Compatible-Devices/dp/B07CHNJ1QN

```
connect the MCU 3.3V RAIL to the DSD TECH VCC PIN
connect the MCU GND RAIL to the DSD TECH GND PIN
connect the MCU A3 PIN to the DSD TECH TXD PIN
connect the MCU A2 PIN to the DSD TECH RXD PIN
```

SSD1306 Display
https://www.amazon.com/Hosyond-Display-Self-Luminous-Compatible-Raspberry/dp/B09T6SJBV5

```
connect the MCU 3.3V RAIL to the SSD1306 VCC PIN
connect the MCU GND RAIL to the SSD1306 GND PIN
connect the MCU B8 PIN to the SSD1306 SCL PIN
connect the MCU B9 PIN to the SSD1306 SDA PIN
```

8x8 WS2182 NeoPixel Array
https://www.amazon.com/Tiabiaya-LED-Panel-CJMCU-8X8-Module/dp/B0BLGN88NH

```
connect the MCU 3.3V RAIL to the WS2812 +5V PIN
connect the MCU GND RAIL to the WS2812 GND PIN
connect the MCU A8 PIN to the WS2812 DIN PIN
```

6x6x5mm Momentary Tactile Tact Push Button Switches
https://www.amazon.com/DAOKI-Miniature-Momentary-Tactile-Quality/dp/B01CGMP9GY

```
connect the MCU GND RAIL to the LEFT BUTTON RIGHT PIN
connect the MCU C13 PIN to the LEFT BUTTON LEFT PIN
connect the MCU GND RAIL to the RIGHT BUTTON RIGHT PIN
connect the MCU C14 PIN to the RIGHT BUTTON LEFT PIN
```

Our last step is to download Bluetooth software to connect with our MCU (micro controller or STM32F401CCU).

Android
Serial Bluetooth Terminal
https://play.google.com/store/apps/details?id=de.kai_morich.serial_bluetooth_terminal&hl=en_US&gl=US

iPhone
BLE Serial tiny
https://apps.apple.com/us/app/ble-serial-tiny/id1607862132

Either option when connecting you will connect to the DSD TECH device.

Now let's review our **main.c** file as this is located in the **src** folder.

```
#include <stdio.h>

#include "usart.h"

int main(void)
{
    usart2_init();

    while (1)
    {
        printf("hello, world\r\n");
    }
}
```

When we connect to our MCU through either app explained above we will see, *hello, world*, displayed over and over. Congrats you just setup and built your first embedded C application!

In our next lesson we will debug hello, world using the ARM embedded GDB with OpenOCD to which we will actually connect LIVE to our running MCU!

Chapter 2: Debugging hello, world

Today we debug!

Before we get started, we are going DEEP and I mean DEEP! Please do not get discouraged as I will take you through literally every single step of the binary but we need to start small.

Assembler is not natural to everyone and I have another FREE book and primer on Embedded Assembler below if you feel you need a good primer. PLEASE take the time to read this book if you are new to this so that you can get the full benefit of this book.

https://github.com/mytechnotalent/Embedded-Assembler

First, open a new terminal and run the GDB debug server.

```
openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
```

Second, in another terminal , run the following to start our GDB debug session. Make sure you are in the root dir of project **0x0001_hello_world**.

```
arm-none-eabi-gdb .\.pio\build\genericSTM32F401CC\firmware.bin
```

Once it loads, we need to target our remote server.

```
target remote :3333
```

We need to halt the currently running binary.

```
monitor reset halt
```

Flash memory or what you would think of as a harddrive starts at address 0x08000000. Let’s look at what is at the beginning of flash memory.

```
(gdb) x/20i 0x08000000
```

What we just did is say, give me 20 instructions at address 0x08000000 or the base of flash memory.

```
   0x8000000:   movs    r0, r0
   0x8000002:   movs    r0, #1
   0x8000004:   lsrs    r5, r3, #17
   0x8000006:   lsrs    r0, r0, #32
   0x8000008:   lsrs    r5, r5, #18
   0x800000a:   lsrs    r0, r0, #32
   0x800000c:   lsrs    r5, r5, #18
   0x800000e:   lsrs    r0, r0, #32
   0x8000010:   lsrs    r5, r5, #18
   0x8000012:   lsrs    r0, r0, #32
   0x8000014:   lsrs    r5, r5, #18
   0x8000016:   lsrs    r0, r0, #32
   0x8000018:   lsrs    r5, r5, #18
   0x800001a:   lsrs    r0, r0, #32
   0x800001c:   movs    r0, r0
   0x800001e:   movs    r0, r0
   0x8000020:   movs    r0, r0
   0x8000022:   movs    r0, r0
   0x8000024:   movs    r0, r0
   0x8000026:   movs    r0, r0
```

Let’s break this down…

```
   0x8000000:   movs    r0, r0
```

This instruction moves the value in register r0 to itself. It's essentially a no-op (no operation), commonly used as a placeholder or for alignment purposes.

```
   0x8000002:   movs    r0, #1
```

This instruction moves the immediate value 1 into register r0. This sets the value of r0 to 1.

Now let’s take a look at what is inside the flash memory location of 0x08000004.

```
(gdb) x/4x 0x08000004
0x8000004:      0x5d    0x0c    0x00    0x08
```

We are using what we refer to as a little endian machine meaning that the byte order from flash or sram is reversed when being placed into a register for execution. This simply means this address of flash at 0x08000004 is holding another flash memory location of 0x08000c5d.

So we see there are 4 bytes so let’s examine what is at the flash memory location of 0x08000008.

```
(gdb) x/4x 0x08000008
0x8000008:      0xad    0x0c    0x00    0x08
```

So we see another memory location of 0x08000cad.

If we continue to look into the next several bytes we see either 0x00 into each byte of flash or a repeated move of 0x08000cad along with an occasional 0xad.

Now that we have gathered some basic information let’s examine 1000 bytes and look for our first push which would indicate the beginning of the **.text** section of the binary.

```
(gdb) x/1000i 0x08000000
…
   0x8000194:   push    {r4, lr}
…
```

Here we see our push instruction. So know we have some more information about our binary and know clearly this is where the **.text** section starts.

Let’s take a deeper look when we loaded GDB.

```
(gdb) target remote :3333
Remote debugging using :3333
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x00000000 in ?? ()
(gdb) monitor reset halt
Unable to match requested speed 2000 kHz, using 1800 kHz
Unable to match requested speed 2000 kHz, using 1800 kHz
adapter speed: 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000c78 msp: 0x20010000
```

We see that the pc or program counter is set to 0x08000c78. Let’s examine what is going on at this address which is currently where the binary is at.

```
(gdb) x/24i 0x08000c78
   0x8000c78:   ldr.w   sp, [pc, #52]   ; 0x8000cb0
   0x8000c7c:   ldr     r0, [pc, #52]   ; (0x8000cb4)
   0x8000c7e:   ldr     r1, [pc, #56]   ; (0x8000cb8)
   0x8000c80:   ldr     r2, [pc, #56]   ; (0x8000cbc)
   0x8000c82:   movs    r3, #0
   0x8000c84:   b.n     0x8000c8c
   0x8000c86:   ldr     r4, [r2, r3]
   0x8000c88:   str     r4, [r0, r3]
   0x8000c8a:   adds    r3, #4
   0x8000c8c:   adds    r4, r0, r3
   0x8000c8e:   cmp     r4, r1
   0x8000c90:   bcc.n   0x8000c86
   0x8000c92:   ldr     r2, [pc, #44]   ; (0x8000cc0)
   0x8000c94:   ldr     r4, [pc, #44]   ; (0x8000cc4)
   0x8000c96:   movs    r3, #0
   0x8000c98:   b.n     0x8000c9e
   0x8000c9a:   str     r3, [r2, #0]
   0x8000c9c:   adds    r2, #4
   0x8000c9e:   cmp     r2, r4
   0x8000ca0:   bcc.n   0x8000c9a
   0x8000ca2:   bl      0x8000cca
   0x8000ca6:   bl      0x80002e4
   0x8000caa:   bl      0x80001d4
   0x8000cae:   bx      lr
```

This is the location of the **Reset_Handler** function which is the very first function called when an embedded binary loads.

To locate our main function we must look for the last bl (branch and link) which is a function call before the bx lr (branch and exchange link register).

Let’s examine our **main** function.

```
(gdb) x/5i 0x080001d4
   0x80001d4:   push    {r3, lr}
   0x80001d6:   bl      0x80001f4
   0x80001da:   ldr     r0, [pc, #8]    ; (0x80001e4)
   0x80001dc:   bl      0x80003f4
   0x80001e0:   b.n     0x80001da
```

The bl 0x080001f4 is the call to our **usart2_init** function which is library code I wrote to communicate with the Bluetooth UART driver. We will not get into full embedded reversing until we have a firm understanding of Embedded C much later in this book.

We then ldr r0, [pc, #8] which we can see is loading the effective address of what is INSIDE the flash memory address of 0x080001e4 into the r0 register.

The instruction ldr r0, [pc, #8] means load the contents of the effective address of pc + 8 which is 0x080001e4.

Let’s now examine what is inside this address.

```
(gdb) x/s *0x080001e4
0x8000ce4:      "hello, world\r"
```

This says examine the string pointed to inside 0x080001e4. We see our hello, world string!

This has been quite a bit of information but please take the time and work through this several times and I again encourage you to read the FREE Embedded Assembler if you want a deeper dive into this.

https://github.com/mytechnotalent/Embedded-Assembler

In our next lesson we will hack this hello, world program!