FIRST EDITION – 0.1 release

Kevin Thomas

Forward

Why Assembler?

“There are a good deal of mature and up-and-coming programming languages that abstract away so much of the low-level details that can help develop a project in record time!”

Why Assembler?

“ChatGPT is just going to program everything and we don’t have to worry about all this low-level implementation or really anything for that matter!”

Why Assembler?

The world of IoT is simply immeasurable. IoT devices are literally everywhere and the amount of connected devices are growing at a rate faster than global population.

With the explosion of IoT medical devices, industrial control systems and the immeasurable amount of SMART devices, the priority of understanding embedded architecture is critical for human survival.

We will use a STM32F401CCU6 microcontroller in this course to which I will provide a link if you do not already have such a device.

Below are items you will need for this book.

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

ST-Link V2 Emulator Downloader Programmer
https://www.amazon.com/HiLetgo-Emulator-Downloader-Programmer-STM32F103C8T6/dp/B07SQV6VLZ

NUCLEO-F401RE Development Board (optional for last chapter)
https://www.amazon.com/NUCLEO-F401RE-Nucleo-64-Development-STM32F401RE-connectivity/dp/B07JYBPWN4

HiLetgo ULN2003 Stepper Motor (optional for last chapter)
https://www.amazon.com/HiLetgo-ULN2003-28BYJ-48-Stepper-4-phase/dp/B00LPK0E5A

Dtech USB to TTL Serial Cable (optional for last chapter)
https://www.amazon.com/Serial-Adapter-Signal-Prolific-Windows/dp/B07R8BQYW1

DSD TECH HM-11 Bluetooth 4.0 BLE Module (optional for last chapter)
https://www.amazon.com/DSD-TECH-Bluetooth-Compatible-Devices/dp/B07CHNJ1QN

Electronics Soldering Iron Kit
https://www.amazon.com/Electronics-Adjustable-Temperature-Controlled-Thermostatic/dp/B0B28JQ95M

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

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

5mm LED Light Assorted Kit
https://www.amazon.com/Gikfun-Assorted-Arduino-100pcs-EK8437/dp/B01ER728F6

100 OHM Resistors
https://www.amazon.com/EDGELEC-Resistor-Tolerance-Multiple-Resistance/dp/B07QG1VL1Q

6x6x5mm Momentary Tactile Tact Push Button Switches
https://www.amazon.com/QTEATAK-Momentary-Tactile-Button-Switch/dp/B07VSNN9S2

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

ESP8266 ESP-01 Serial WiFi Wireless Transceiver
https://www.amazon.com/HiLetgo-Wireless-Transceiver-Development-Compatible/dp/B010N1ROQS

If you need a primer on Assembler, download the **Assembler-Primer.pdf** document within the repo below.

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

Let’s begin...

Table Of Contents

Chapter 1: Toolchain

We first need to have a toolchain to which to develop our microcontroller software. The links below are for Windows-based operating systems. If you are on MAC or Linux you can simply brew install or apt-get the same applications.

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

Let’s download OpenOCD.

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

Let’s download VIM.

https://www.vim.org/download.php

Once installed, let’s create a simple project. It is not critical that you understand how this simple program works but only that it compiles as this will be a very long journey.

I want to emphasize, this chapter is ONLY about ensuring the toolchain works. It will take several chapters to dive into what is exactly happening but first lets make sure we are functioning as expected.

```
mkdir stm32f401ccu6-projects
cd stm32f401ccu6-projects
mkdir 0x0001-template
cd 0x0001-template
vim main.s
```

Type in the following code and save as **main.s** and if you are unfamiliar with VIM please watch this video.
https://youtu.be/ggSyF1SVFr4

```
/**
 * FILE: main.s
 *
 * DESCRIPTION:
 * This file contains the assembly code for a boilerplate firmware
 * utilizing the STM32F401CC6 microcontroller.
 *
 * AUTHOR: Kevin Thomas
 * CREATION DATE: July 2, 2023
 * UPDATE Date: July 2, 2023
 *
 * ASSEMBLE AND LINK w/ SYMBOLS:
 * 1. arm-none-eabi-as -g main.s -o main.o
 * 2. arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
 * 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf verify reset exit"
 * ASSEMBLE AND LINK w/o SYMBOLS:
 * 1. arm-none-eabi-as -g main.s -o main.o
 * 2. arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
 * 3. arm-none-eabi-objcopy -O binary --strip-all main.elf main.bin
 * 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.bin 0x08000000 verify reset exit"
 * DEBUG w/ SYMBOLS:
 * 1. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
 * 2. arm-none-eabi-gdb main.elf
 * 3. target remote :3333
```

```
 * 4. monitor reset halt
 * 5. l
 * DEBUG w/o SYMBOLS:
 * 1. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
 * 2. arm-none-eabi-gdb main.bin
 * 3. target remote :3333
 * 4. monitor reset halt
 * 5. x/8i $pc
 */

.syntax unified
.cpu cortex-m4
.thumb

/**
 * Provide weak aliases for each Exception handler to the Default_Handler.
 * As they are weak aliases, any function with the same name will override
 * this definition.
 */
.macro weak name
  .global \name
  .weak \name
  .thumb_set \name, Default_Handler
  .word \name
.endm

/**
 * The STM32F401CCUx vector table.  Note that the proper constructs
 * must be placed on this to ensure that it ends up at physical address
 * 0x0000.0000.
 */
.global isr_vector
.section .isr_vector, "a"
.type isr_vector, %object
isr_vector:
  .word _estack
  .word Reset_Handler
   weak NMI_Handler
   weak HardFault_Handler
   weak MemManage_Handler
   weak BusFault_Handler
   weak UsageFault_Handler
  .word 0
  .word 0
  .word 0
  .word 0
   weak SVC_Handler
   weak DebugMon_Handler
  .word 0
   weak PendSV_Handler
   weak SysTick_Handler
  .word 0
   weak EXTI16_PVD_IRQHandler                          // EXTI Line 16 interrupt /PVD through EXTI line detection interrupt
   weak TAMP_STAMP_IRQHandler                          // Tamper and TimeStamp interrupts through the EXTI line
   weak EXTI22_RTC_WKUP_IRQHandler                     // EXTI Line 22 interrupt /RTC Wakeup interrupt through the EXTI line
   weak FLASH_IRQHandler                               // FLASH global interrupt
   weak RCC_IRQHandler                                 // RCC global interrupt
   weak EXTI0_IRQHandler                               // EXTI Line0 interrupt
   weak EXTI1_IRQHandler                               // EXTI Line1 interrupt
   weak EXTI2_IRQHandler                               // EXTI Line2 interrupt
   weak EXTI3_IRQHandler                               // EXTI Line3 interrupt
   weak EXTI4_IRQHandler                               // EXTI Line4 interrupt
   weak DMA1_Stream0_IRQHandler                        // DMA1 Stream0 global interrupt
   weak DMA1_Stream1_IRQHandler                        // DMA1 Stream1 global interrupt
   weak DMA1_Stream2_IRQHandler                        // DMA1 Stream2 global interrupt
   weak DMA1_Stream3_IRQHandler                        // DMA1 Stream3 global interrupt
   weak DMA1_Stream4_IRQHandler                        // DMA1 Stream4 global interrupt
   weak DMA1_Stream5_IRQHandler                        // DMA1 Stream5 global interrupt
   weak DMA1_Stream6_IRQHandler                        // DMA1 Stream6 global interrupt
   weak ADC_IRQHandler                                 // ADC1 global interrupt
  .word 0                                              // Reserved
  .word 0                                              // Reserved
  .word 0                                              // Reserved
  .word 0                                              // Reserved
   weak EXTI9_5_IRQHandler                             // EXTI Line[9:5] interrupts
   weak TIM1_BRK_TIM9_IRQHandle                        // TIM1 Break interrupt and TIM9 global interrupt
   weak TIM1_UP_TIM10_IRQHandler                       // TIM1 Update interrupt and TIM10 global interrupt
   weak TIM1_TRG_COM_TIM11_IRQHandler                  // TIM1 Trigger and Commutation interrupts and TIM11 global interrupt
   weak TIM1_CC_IRQHandler                             // TIM1 Capture Compare interrupt
   weak TIM2_IRQHandler                                // TIM2 global interrupt
   weak TIM3_IRQHandler                                // TIM3 global interrupt
   weak TIM4_IRQHandler                                // TIM4 global interrupt
   weak I2C1_EV_IRQHandler                             // I2C1 event interrupt
   weak I2C1_ER_IRQHandler                             // I2C1 error interrupt
   weak I2C2_EV_IRQHandler                             // I2C2 event interrupt
   weak I2C2_ER_IRQHandler                             // I2C2 error interrupt
   weak SPI1_IRQHandler                                // SPI1 global interrupt
   weak SPI2_IRQHandler                                // SPI2 global interrupt
   weak USART1_IRQHandler                              // USART1 global interrupt
   weak USART2_IRQHandler                              // USART2 global interrupt
  .word 0                                              // Reserved
   weak EXTI15_10_IRQHandler                           // EXTI Line[15:10] interrupts
   weak EXTI17_RTC_Alarm_IRQHandler                    // EXTI Line 17 interrupt / RTC Alarms (A and B) through EXTI line interrupt
```

```
   weak EXTI18_OTG_FS_WKUP_IRQHandler                        // EXTI Line 18 interrupt / USBUSB OTG FS Wakeup through EXTI line interrupt
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
   weak DMA1_Stream7_IRQHandler                              // DMA1 Stream7 global interrupt
  .word 0                                                    // Reserved
   weak SDIO_IRQHandler                                      // SDIO global interrupt
   weak TIM5_IRQHandler                                      // TIM5 global interrupt
   weak SPI3_IRQHandler                                      // SPI3 global interrupt
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
   weak DMA2_Stream0_IRQHandler                              // DMA2 Stream0 global interrupt
   weak DMA2_Stream1_IRQHandler                              // DMA2 Stream1 global interrupt
   weak DMA2_Stream2_IRQHandler                              // DMA2 Stream2 global interrupt
   weak DMA2_Stream3_IRQHandler                              // DMA2 Stream3 global interrupt
   weak DMA2_Stream4_IRQHandler                              // DMA2 Stream4 global interrupt
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
   weak OTG_FS_IRQHandler                                    // USB On The Go FS global interrupt
   weak DMA2_Stream5_IRQHandler                              // DMA2 Stream5 global interrupt
   weak DMA2_Stream6_IRQHandler                              // DMA2 Stream6 global interrupt
   weak DMA2_Stream7_IRQHandler                              // DMA2 Stream7 global interrupt
   weak USART6_IRQHandler                                    // USART6 global interrupt
   weak I2C3_EV_IRQHandler                                   // I2C3 event interrupt
   weak I2C3_ER_IRQHandler                                   // I2C3 error interrupt
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
  .word 0                                                    // Reserved
   weak SPI4_IRQHandler                                      // SPI4 global interrupt

.section .text

/**
 * @brief  This code is called when processor starts execution.
 *
 *         This is the code that gets called when the processor first
 *         starts execution following a reset event. Only the absolutely
 *         necessary set is performed, after which the application
 *         supplied main() routine is called.
 * @param  None
 * @retval None
 */
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
  LDR   R0, =_estack                                         // load address at end of the stack into R0
  MOV   SP, R0                                               // move address at end of stack into SP
  BL    __start                                              // call function

/**
 * @brief  This code is called when the processor receives and unexpected interrupt.
 *
 *         This is the code that gets called when the processor receives an
 *         unexpected interrupt.  This simply enters an infinite loop, preserving
 *         the system state for examination by a debugger.
 *
 * @param  None
 * @retval None
 */
.type Default_Handler, %function
.global Default_Handler
Default_Handler:
  BKPT                                                       // set processor into debug state
  B.N   Default_Handler                                      // call function, force thumb state

/**
 * @brief  Entry point for initialization and setup of specific functions.
 *
 *         This function is the entry point for initializing and setting up specific functions.
 *         It calls other functions to enable certain features and then enters a loop for further execution.
 *
 * @param  None
 * @retval None
 */
.type __start, %function
__start:
  NOP                                                        // no operation instruction
  B    .                                                     // branch infinite loop
```

Let’s code up our linker script and save it as **stm32f401ccux.ld** filename.

```
/**
 * FILE: stm32f401ccux.ld
 *
 * DESCRIPTION:
 * This file contains the linker script
 * utilizing the STM32F401CC6 microcontroller.
 *
 * AUTHOR: Kevin Thomas
 * CREATION DATE: July 2, 2023
 * UPDATE Date: July 2, 2023
 */


MEMORY
{
  FLASH : ORIGIN = 0x08000000, LENGTH = 256K
  SRAM  : ORIGIN = 0x20000000, LENGTH = 64K
}


SECTIONS
{
  .isr_vector :
  {
    *(.isr_vector)
  } >FLASH
  .text :
  {
    *(.text)
  } >FLASH
  .data (NOLOAD) :
  {
    . = . + 0x400;
          _estack = .;
          *(.data)
  } >SRAM
}
```

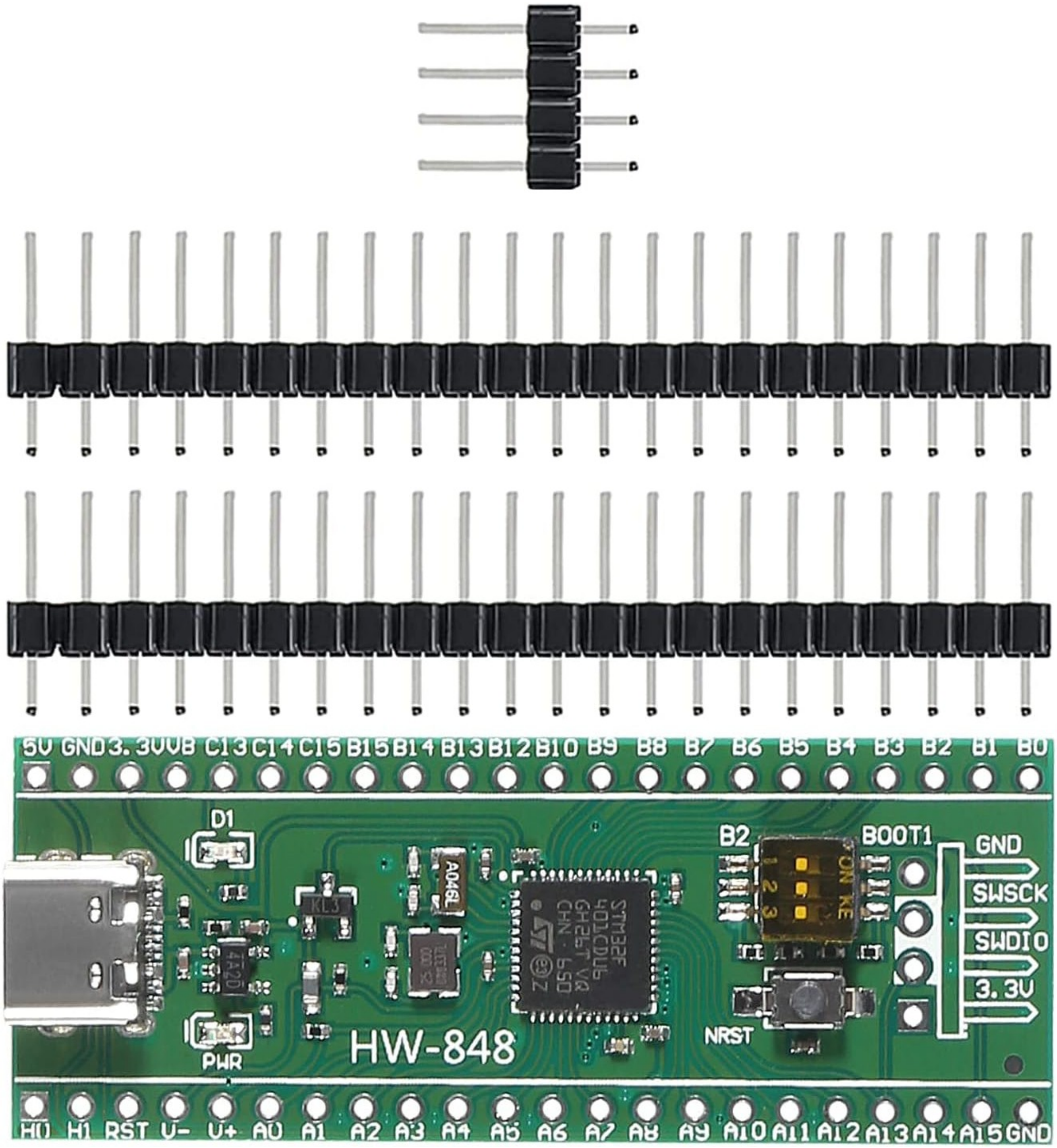
Let’s assemble our simple source code.

```
arm-none-eabi-as -g main.s -o main.o
```

The next step is to link the object code to a ELF binary.

```
arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
```

Now it is time to hook up our ST-Link V2 Emulator Downloader Programmer. Download driver here https://www.st.com/en/development-tools/stsw-link009.html if you do not have STM32CubeIDE installed.



After soldering all of the pins, we see 4 pins on the right of the device. First, connect the GND pin to the GND pin on the ST-Link. Second, connect the SWSCK pin to the SWSCK pin on the ST-Link. Third, connect the SWDIO pin to the SWDIO pin on the ST-Link. Finally, connect the 3.3V pin to the 3.3V pin on the ST-Link.

Now it is time to flash our program to the device.

```
openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf verify reset exit"
```

To ensure our firmware is successful, let’s examine it in our debugger.

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

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

Second, in our original terminal, run the following to start our GDB debug session.

```
arm-none-eabi-gdb main.elf
```

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

```
target remote :3333
```

We next need to halt the currently running binary.

```
monitor reset halt
```

We can now see our source code by typing the following.

```
l
```

```
1       /**
2        * FILE: main.s
3        *
4        * DESCRIPTION:
5        * This file contains the assembly code for a boilerplate firmware
6        * utilizing the STM32F401CC6 microcontroller.
7        *
8        * AUTHOR: Kevin Thomas
9        * CREATION DATE: July 2, 2023
10       * UPDATE Date: July 2, 2023
(gdb)
```

We should now see our source code. At this point we can step into the code, instruction-by-instruction.

```
si
```

```
Reset_Handler () at main.s:177
177       MOV  SP, R0
```

After a second step, we see that we entering into our __start function.

```
si
```

```
Reset_Handler () at main.s:178
178       BL   __start
```

After a third step, we hit our NOP instruction.

```
si
```

```
__start () at main.s:208
208       NOP                                           // no operation instruction
```

After a fourth step, we are in an infinite loop.

```
si
```

```
209       B    .                                        // branch infinite loop
```

To exit the debugger simply type the following.

```
q
```

You can then CTRL-C the GDB server.

In our next lesson we will dive into architecture basics.

Chapter 2: Architecture Basics

Now that we have as working template, it’s time to dive into some architecture basics of the STM32F401CCU6.

There are two primary manuals we will use when developing software which are the datasheet and the reference manual. Both documents are included in the GitHub repo.

If we open up the datasheet, we first want to search for the memory map which is on page 50.

The first thing we need to understand is that this MCU or microcontroller utilizes a ARM 32-bit thumb architecture.

The ARM 32-bit Thumb architecture is an instruction set architecture (ISA) developed by ARM Holdings. It is designed to be a compact and efficient instruction set primarily targeted for use in low-power and resource-constrained embedded systems. The name "Thumb" refers to the reduced instruction set's goal of fitting 16-bit instructions (thumb instructions) to reduce code size while still retaining good performance.

Key features of the ARM 32-bit Thumb architecture include:

16-bit Thumb Instructions: Thumb instructions are 16 bits long, which is half the size of the standard 32-bit ARM instructions. This reduction in instruction size leads to smaller memory footprints, making it ideal for systems with limited memory capacity.

Subset of ARM ISA: The Thumb instruction set is a subset of the full 32-bit ARM instruction set (referred to as "ARM" or "ARM32"). While some instructions have been simplified or removed, most of the essential instructions for efficient code execution remain.

Efficient Execution: Despite the instruction size reduction, the Thumb architecture maintains good performance due to various optimizations and trade-offs in the instruction design. Thumb instructions can still access 32-bit registers, making it possible to perform 32-bit arithmetic and logical operations.

Interworking Support: ARM processors that support the Thumb architecture typically have the ability to switch between Thumb and ARM instruction sets during runtime. This feature allows seamless integration of Thumb code with existing ARM code when necessary.

Code Density: The primary advantage of the Thumb architecture is its improved code density. Since Thumb instructions are smaller, more instructions can fit into the same memory space compared to full-sized ARM instructions. This is particularly beneficial for memory-constrained embedded systems.

Limited Features: While the Thumb instruction set provides many essential instructions, some advanced features available in the full ARM instruction set may not be available in Thumb mode. This trade-off ensures that the architecture remains compact and efficient.

The Thumb architecture is particularly popular in the ARM Cortex-M series of microcontrollers, which are widely used in various embedded applications, including IoT devices, consumer electronics, automotive systems, and more. The Cortex-M series processors often feature low power consumption, cost-effectiveness, and are optimized for real-time and resource-constrained environments, making the Thumb architecture a preferred choice in such scenarios.

On the far left of the document you notice the entire memory space starting from 0x00000000 to 0xFFFFFFFF.

This represents the maximum space you have to work with and not all of it is available on the MCU.

The first thing to realize is that the maximum address is 0xFFFFFFFF or 4,294,967,295 in decimal. This value is often used in computing as the maximum unsigned 32-bit integer. It is equivalent to 2^32 - 1, where the "1" comes from the lowest bit, and the other 32 bits are all set to "1."

On page 51 of the datasheet, you will see the register boundary addresses. So, for example, when the processor is reading or writing to 0x40020000 it is referring to GPIOA and more specifically GPIOx_MODER which is on page 158 of the reference manual to which the GPIOx_MODER register is at offset 0x00 so it lives literally at 0x40020000.

If you were to write into 0x40020000 you would in this instance configure the I/O direction mode such as input, output, etc.

At this point we want to review what we refer to as the block diagram of our MCU. If you turn to page 14 of the datasheet you will see that our CPU has a maximum speed of 84 MHz. Keep in mind that without making a variety of configurations, it will default at 16 MHz.

We first notice on the top that there are 3 buses. There exists a D-BUS, I-BUS and S-BUS.

The data bus or D-BUS, handles communication between the processor and FLASH regarding data within the binary.

The instruction bus or I-BUS, handles communication between the processor and FLASH regarding instructions within the binary.

Let’s break down some differences between the two.

Instructions are a set of commands or operations that direct the CPU on how to perform tasks. They represent the program's logic and tell the CPU what operations to execute and in what sequence. Instructions are encoded in binary form (machine code) and are stored in the memory of the computer as a sequence of binary digits (0s and 1s).

When the CPU executes a program, it fetches instructions from memory one by one, decodes them to understand their meaning, and then executes the corresponding operation. Instructions can include arithmetic and logical operations, control flow instructions (e.g., conditional jumps and loops), memory access operations, and other specialized operations based on the CPU's instruction set architecture (ISA).

For example, an instruction might tell the CPU to add two numbers together, load a value from memory into a register, or jump to a different part of the program based on a condition.

Data represents the information processed by the instructions. It can be numeric values, characters, strings, images, sound, or any other type of information. Data is also stored in the memory of the computer, separate from the program's instructions.

The CPU manipulates data according to the instructions it executes. For example, if an instruction involves adding two numbers, the CPU will fetch the data (the two numbers) from memory, perform the addition, and store the result back in memory or a register.

Data can be categorized into different types, such as integers, floating-point numbers, characters, booleans, and more. The way data is represented and manipulated depends on the data types and the operations specified by the instructions.

The processor will fetch the instruction from FLASH on the I-BUS and the processor will use the D-BUS to read the data on the FLASH.

FLASH is made up of a vector table at the base of memory followed by constant data and finally instructions.

FLASH is connected to the MCU throug the FLASH I/F or controller.

The system bus or S-BUS will allow communication between the MCU and the various peripherals over the AHB and APB buses.

Inside the ARM Cortex-M4 Technical Reference Manual we see on page 24 and 25 the following.

ICode memory interface

Instruction fetches from Code memory space, 0x00000000 to 0x1FFFFFFC, are performed over the 32-bit AHB-Lite bus.

The Debugger cannot access this interface. All fetches are word-wide. The number of instructions fetched per word depends on the code running and the alignment of the code in memory.

DCode memory interface

Data and debug accesses to Code memory space, 0x00000000 to 0x1FFFFFFF, are performed over the 32-bit AHB-Lite bus.

The Code memory space available is dependent on the implementation. Core data accesses have a higher priority than debug accesses on this bus. This means that debug accesses are waited until core accesses have completed when there are simultaneous core and debug access to this bus.

Control logic in this interface converts unaligned data and debug accesses into two or three aligned accesses, depending on the size and alignment of the unaligned access. This stalls any subsequent data or debug access until the unaligned access has completed.

Note: ARM strongly recommends that any external arbitration between the ICode and DCode AHB bus interfaces ensures that DCode has a higher priority than ICode.

System interface

Instruction fetches and data and debug accesses to address ranges 0x20000000 to 0xDFFFFFFF and 0xE0100000 to 0xFFFFFFFF are performed over the 32-bit AHB-Lite bus. For simultaneous accesses to the 32-bit AHB-Lite bus, the arbitration order in decreasing priority is:

- Data accesses.
- Instruction and vector fetches.
- Debug.

The system bus interface contains control logic to handle unaligned accesses, FPB remapped accesses, bit-band accesses, and pipelined instruction fetches.

To summarize…

If the instructions are present in between memory locations 0x00000000 to 0x1FFFFFFFC then the MCU will fetch the instructions over the I-BUS.

If the data is present in between 0x00000000 to 0x1FFFFFFFF then the MCU will fetch the data over the D-BUS.

If the instructions are present outside of 0x00000000 to 0x1FFFFFFFC then the MCU will fetch the instructions over the S-BUS.

The processor can fetch instructions as well as data from SRAM as they have two buses which are I-BUS and D-BUS.

The S-BUS can only interface with one peripheral address at a time.

Now it is time to understand the embedded flash within the reference manual. If we turn to page 45 we see main memory starting at sector 0 at 0x08000000 to 0x08003FFF.

We see there are also sectors 2, 3, 4 and 5. In our MCU we do not have sectors 6 or 7.

In our next lesson we will cover the vector table.

Chapter 3: Vector Table

Now that we have the basics of how the MCU is designed we can start to look deeper into how the code works and how it is structured.

Let’s turn to page 202 of our reference manual.

At the very base of memory is what we refer to as the MSP or master stack pointer. It is 4 bytes long and it is were we first initialize the stack.

We see at address 0x00000000 that it is reserved this is where we put our MSP or master stack pointer in our code.

The very next thing we see is the address of the Reset Handler which is at 0x00000004.

All in all we have the following.

* 85 IRQ's
* 15 system exceptions
* master stack pointer
* 85+15+1 = 101 * 4 = 404 bytes or 0x194 bytes

We can prove this by running the following.

```
arm-none-eabi-objdump -h main.o

main.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000014  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000048  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000048  2**0
                  ALLOC
  3 .isr_vector   00000194  00000000  00000000  00000048  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  4 .debug_line   00000044  00000000  00000000  000001dc  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  5 .debug_info   00000026  00000000  00000000  00000220  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00000014  00000000  00000000  00000246  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 00000020  00000000  00000000  00000260  2**3
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000053  00000000  00000000  00000280  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .ARM.attributes 00000021  00000000  00000000  000002d3  2**0
                  CONTENTS, READONLY
```

This table handles all of the various handlers to which the reset handler will be the first thing executed after the MSP gets set and the other handlers will handle when something goes wrong.

They are weakly aliased to the default handler except for when we explicitly code up a function like we do with the reset handler.

The reset of the values correspond to interrupt handling such as pressing a button and it interrupting the processor so you can avoid having what we call a blocking call.

Imagine you have a program where you are in a loop and you iterate through a number of code items. With an interrupt you can have code execute without being part of the main code with is extremely powerful.

The role of the reset handler is to set the address at the end of stack and place that into one of our general purpose registers that we will cover in another lesson. Then we take that value and move it into the SP or stack pointer register and then we branch and link to __start where our application begins.

In our next lesson we will cover the linker script.

Chapter 4: Linker Script

We are making some progress on our journey and now it is time to understand how the linker script works.

When we assemble our instructions it creates what we refer to as a relocatable object file. What this means is all of the addresses are mapped to 0x00000000. The job of the linker is to link the various object files, in our case just one, to actual addresses within our flash.

Let's review our linker script.

```
/**
 * FILE: stm32f401ccux.ld
 *
 * DESCRIPTION:
 * This file contains the linker script
 * utilizing the STM32F401CC6 microcontroller.
 *
 * AUTHOR: Kevin Thomas
 * CREATION DATE: July 2, 2023
 * UPDATE Date: July 2, 2023
 */

MEMORY
{
  FLASH : ORIGIN = 0x08000000, LENGTH = 256K
  SRAM  : ORIGIN = 0x20000000, LENGTH = 64K
}

SECTIONS
{
  .isr_vector :
  {
    *(.isr_vector)
  } >FLASH
  .text :
  {
    *(.text)
  } >FLASH
  .data (NOLOAD) :
  {
    . = . + 0x400;
          _estack = .;
          *(.data)
  } >SRAM
}
```

We first notice that our FLASH is 256,000 bytes. We also see that FLASH is going to get mapped to address 0x08000000 and we see SRAM at a length of 64,000 bytes to which we are going to map to address 0x20000000.

We see that at the base of FLASH we map the vector table to 0x08000000 as we know that the object file is mapped to 0x00000000 so now it will link to 0x08000000.

We also see a value here called _estack which is 0x20000400 which is the end of stack as well.

In our next lesson we will dive ELF file analysis.

Chapter 5: ELF File Analysis

As we are peeling away the layers to understand how our microcontroller works, we now are at the stage where we can start examining what the binary is made up of.

First, what is ELF?

The ELF (Executable and Linkable Format) file format is a widely used binary file format that is used for executables, object code, shared libraries, and even core dumps. It is designed to be platform-independent, making it suitable for a wide range of architectures, including microcontrollers like the STM32F401CCU6. The ELF format allows the microcontroller's firmware to be stored in a structured and standardized way, enabling easy integration with various tools and platforms.

The ELF file format consists of several sections and headers, each serving a specific purpose. Let's go through some of the key components of the ELF file format as they pertain to the STM32F401CCU6 microcontroller:

ELF Header: The ELF header is located at the beginning of the file and contains general information about the file, such as the target architecture (e.g., ARM), the type of the file (executable, object file, etc.), the entry point address (the starting point of the program), and various other flags and offsets.

Program Header Table: The program header table provides information about the different loadable segments of the binary. In the case of microcontrollers like the STM32F401CCU6, this typically includes sections for code, data, and possibly other sections like initialization routines or interrupt vectors.

Section Header Table: The section header table contains information about various sections in the binary, such as code sections, data sections, symbol table, and debug information. Each section has a specific purpose in the program's execution, and the section header table helps the loader and debugger to navigate and understand the file's structure.

Code and Data Sections: These sections contain the actual instructions and data that make up the firmware or program. Code sections hold the machine instructions that the microcontroller's CPU executes, while data sections contain initialized and uninitialized data used by the program.

Symbol Table: The symbol table contains information about the names and addresses of functions, variables, and other symbols used in the program. It is vital for debugging and resolving external references during the linking process.

Relocation Information: Relocation information provides details on how to modify the binary's addresses during the linking process to accommodate the actual memory layout of the microcontroller.

Debug Information: Optional debugging information is often included in the ELF file to aid in debugging the program using tools like gdb (GNU Debugger).

The first tool we will use is to display the contents of the section headers.

```
arm-none-eabi-objdump -h main.o

main.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000014  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000048  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000048  2**0
                  ALLOC
  3 .isr_vector   00000194  00000000  00000000  00000048  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  4 .debug_line   00000044  00000000  00000000  000001dc  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  5 .debug_info   00000026  00000000  00000000  00000220  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00000014  00000000  00000000  00000246  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 00000020  00000000  00000000  00000260  2**3
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000053  00000000  00000000  00000280  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .ARM.attributes 00000021  00000000  00000000  000002d3  2**0
                  CONTENTS, READONLY
```

This should look familiar as we reviewed this briefly in a prior chapter. Let’s break this down.

.text Section:

Size: 0x14 bytes (20 bytes)
Virtual Memory Address (VMA): 0x00000000
Load Memory Address (LMA): 0x00000000
File Offset: 0x00000034
Alignment: 2^2 (4 bytes)
Attributes: CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE

Explanation: The .text section contains the machine instructions (code) of the program. The size of this section is 20 bytes. It will be loaded into memory starting from address 0x00000000 (VMA and LMA), and its content is present at file offset 0x00000034 within the ELF file. The section is marked as readable and executable (READONLY, CODE) and will be relocated during the linking process.

.data Section:

Size: 0x00 bytes (0 bytes)
VMA: 0x00000000
LMA: 0x00000000
File Offset: 0x00000048
Alignment: 2^0 (1 byte)
Attributes: CONTENTS, ALLOC, LOAD, DATA
Explanation: The .data section contains initialized data used by the program. It has a size of 0 bytes, indicating that there are no explicitly initialized data items in this section. The section will be loaded into memory starting from address 0x00000000 and is located at file offset 0x00000048 within the ELF file. This section is marked as readable and writable (CONTENTS, ALLOC, LOAD, DATA).

.bss Section:

Size: 0x00 bytes (0 bytes)
VMA: 0x00000000
LMA: 0x00000000
File Offset: 0x00000048
Alignment: 2^0 (1 byte)
Attributes: ALLOC
Explanation: The .bss section contains uninitialized data used by the program. Similar to the .data section, it has a size of 0 bytes, indicating that there are no explicitly uninitialized data items in this section. The section will be allocated memory but not loaded from the file. Instead, it will be initialized to zero during program startup. The section is marked with the ALLOC attribute.

.isr_vector Section:

Size: 0x194 bytes (404 bytes)
VMA: 0x00000000
LMA: 0x00000000
File Offset: 0x00000048
Alignment: 2^0 (1 byte)
Attributes: CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA

Explanation: The .isr_vector section contains the interrupt vector table, which holds the addresses of various interrupt service routines (ISRs). The size of this section is 404 bytes. Like other data sections, it will be loaded into memory, and its content is present at file offset 0x00000048. It is marked as readable and non-modifiable (READONLY) and will be relocated during the linking process.

.debug_line Section:

Size: 0x44 bytes (68 bytes)
VMA: 0x00000000
LMA: 0x00000000
File Offset: 0x000001dc
Alignment: 2^0 (1 byte)
Attributes: CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
Explanation: The .debug_line section contains debugging information related to source code line numbers and mapping between source code and generated machine code. This information is useful for debugging purposes. The section is marked as readable (CONTENTS, READONLY) and contains relocatable entries (RELOC) and debugging data (DEBUGGING, OCTETS).

.debug_info Section:

Size: 0x26 bytes (38 bytes)
VMA: 0x00000000
LMA: 0x00000000
File Offset: 0x00000220
Alignment: 2^0 (1 byte)
Attributes: CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
Explanation: The .debug_info section contains debugging information about program entities such as variables, types, and functions. This information is used during debugging sessions to provide detailed information about the program's data structures and functions. The section is marked as readable (CONTENTS, READONLY) and contains relocatable entries (RELOC) and debugging data (DEBUGGING, OCTETS).

.debug_abbrev Section:

Size: 0x14 bytes (20 bytes)
VMA: 0x00000000
LMA: 0x00000000
File Offset: 0x00000246
Alignment: 2^0 (1 byte)
Attributes: CONTENTS, READONLY, DEBUGGING, OCTETS

Explanation: The .debug_abbrev section contains abbreviation tables used in debugging information to represent common data structures compactly. The section is marked as readable (CONTENTS, READONLY) and contains debugging data (DEBUGGING, OCTETS).

.debug_aranges Section:

Size: 0x20 bytes (32 bytes)
VMA: 0x00000000
LMA: 0x00000000
File Offset: 0x00000260
Alignment: 2^3 (8 bytes)
Attributes: CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
Explanation: The .debug_aranges section contains address range information for debugging. It helps to map machine code addresses to source code line numbers during debugging sessions. The section is marked as readable (CONTENTS, READONLY) and contains relocatable entries (RELOC) and debugging data (DEBUGGING, OCTETS).

.debug_str Section:

Size: 0x53 bytes (83 bytes)
VMA: 0x00000000
LMA: 0x00000000
File Offset: 0x00000280
Alignment: 2^0 (1 byte)
Attributes: CONTENTS, READONLY, DEBUGGING, OCTETS
Explanation: The .debug_str section contains debug information strings, such as variable and function names, used during debugging. The section is marked as readable (CONTENTS, READONLY) and contains debugging data (DEBUGGING, OCTETS).

.ARM.attributes Section:

Size: 0x21 bytes (33 bytes)
VMA: 0x00000000
LMA: 0x00000000
File Offset: 0x000002d3
Alignment: 2^0 (1 byte)
Attributes: CONTENTS, READONLY
Explanation: The .ARM.attributes section contains attributes specific to the ARM architecture, describing various characteristics of the binary. The section is marked as readable (CONTENTS, READONLY).

Overall, this output provides detailed information about the different sections present in the main.o object file, which will be used in the linking process to generate the final executable or firmware for the STM32F401CCU6 microcontroller.

The next tool we will look at will display assembler contents of all sections.

```
arm-none-eabi-objdump -D main.o | less

main.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <Reset_Handler>:
   0:   4803            ldr     r0, [pc, #12]   ; (10 <__start+0x4>)
   2:   4685            mov     sp, r0
   4:   f000 f802       bl      c <__start>

00000008 <Default_Handler>:
   8:   be00            bkpt    0x0000
   a:   e7fd            b.n     8 <Default_Handler>

0000000c <__start>:
   c:   bf00            nop
   e:   e7fe            b.n     e <__start+0x2>
  10:   00000000        andeq   r0, r0, r0

Disassembly of section .isr_vector:

00000000 <isr_vector>:
        ...

Disassembly of section .debug_line:

00000000 <.debug_line>:
   0:   00000040        andeq   r0, r0, r0, asr #32
   4:   001d0003        andseq  r0, sp, r3
   8:   01020000        mrseq   r0, (UNDEF: 2)
   c:   000d0efb        strdeq  r0, [sp], -fp
  10:   01010101        tsteq   r1, r1, lsl #2
  14:   01000000        mrseq   r0, (UNDEF: 0)
  18:   00010000        andeq   r0, r1, r0
  1c:   6e69616d        powvsez f6, f1, #5.0
  20:   0000732e        andeq   r7, r0, lr, lsr #6
  24:   00000000        andeq   r0, r0, r0
  28:   00000205        andeq   r0, r0, r5, lsl #4
  2c:   b0030000        andlt   r0, r3, r0
  30:   21210101                        ; <UNDEFINED> instruction: 0x21210101
  34:   212e0f03                        ; <UNDEFINED> instruction: 0x212e0f03
  38:   21200d03                        ; <UNDEFINED> instruction: 0x21200d03
  3c:   02206003        eoreq   r6, r0, #3
  40:   01010002        tsteq   r1, r2

Disassembly of section .debug_info:

00000000 <.debug_info>:
   0:   00000022        andeq   r0, r0, r2, lsr #32
   4:   00000002        andeq   r0, r0, r2
   8:   01040000        mrseq   r0, (UNDEF: 4)
        ...
  14:   00000014        andeq   r0, r0, r4, lsl r0
  18:   00000000        andeq   r0, r0, r0
  1c:   00000007        andeq   r0, r0, r7
  20:   00000044        andeq   r0, r0, r4, asr #32
  24:   Address 0x24 is out of bounds.
```

```
Disassembly of section .debug_abbrev:

00000000 <.debug_abbrev>:
   0:   10001101        andne   r1, r0, r1, lsl #2
   4:   12011106        andne   r1, r1, #-2147483647    ; 0x80000001
   8:   1b0e0301        blne    380c14 <__start+0x380c08>
   c:   130e250e        movwne  r2, #58638      ; 0xe50e
  10:   00000005        andeq   r0, r0, r5

Disassembly of section .debug_aranges:

00000000 <.debug_aranges>:
   0:   0000001c        andeq   r0, r0, ip, lsl r0
   4:   00000002        andeq   r0, r0, r2
   8:   00040000        andeq   r0, r4, r0
        ...
  14:   00000014        andeq   r0, r0, r4, lsl r0
        ...

Disassembly of section .debug_str:

00000000 <.debug_str>:
   0:   6e69616d        powvsez f6, f1, #5.0
   4:   4300732e        movwmi  r7, #814        ; 0x32e
   8:   73555c3a        cmpvc   r5, #14848      ; 0x3a00
   c:   5c737265        lfmpl   f7, 2, [r3], #-404      ; 0xfffffe6c
  10:   6574796d        ldrbvs  r7, [r4, #-2413]!       ; 0xfffff693
  14:   74735c63        ldrbtvc r5, [r3], #-3171        ; 0xfffff39d
  18:   6632336d        ldrtvs  r3, [r2], -sp, ror #6
  1c:   63313034        teqvs   r1, #52 ; 0x34
  20:   2d367563        cfldr32cs       mvfx7, [r6, #-396]!     ; 0xfffffe74
  24:   6a6f7270        bvs     1bdc9ec <__start+0x1bdc9e0>
  28:   73746365        cmnvc   r4, #-1811939327        ; 0x94000001
  2c:   6f6c635c        svcvs   0x006c635c
  30:   5c746573        cfldr64pl       mvdx6, [r4], #-460      ; 0xfffffe34
  34:   30307830        eorscc  r7, r0, r0, lsr r8
  38:   742d3130        strtvc  r3, [sp], #-304 ; 0xfffffed0
  3c:   6c706d65        ldclvs  13, cr6, [r0], #-404    ; 0xfffffe6c
  40:   00657461        rsbeq   r7, r5, r1, ror #8
  44:   20554e47        subscs  r4, r5, r7, asr #28
  48:   32205341        eorcc   r5, r0, #67108865       ; 0x4000001
  4c:   2e39332e        cdpcs   3, 3, cr3, cr9, cr14, {1}
  50:   Address 0x50 is out of bounds.


Disassembly of section .ARM.attributes:

00000000 <.ARM.attributes>:
   0:   00002041        andeq   r2, r0, r1, asr #32
   4:   61656100        cmnvs   r5, r0, lsl #2
   8:   01006962        tsteq   r0, r2, ror #18
   c:   00000016        andeq   r0, r0, r6, lsl r0
  10:   726f4305        rsbvc   r4, pc, #335544320      ; 0x14000000
  14:   2d786574        cfldr64cs       mvdx6, [r8, #-464]!     ; 0xfffffe30
  18:   0600344d        streq   r3, [r0], -sp, asr #8
  1c:   094d070d        stmdbeq sp, {r0, r2, r3, r8, r9, sl}^
  20:   Address 0x20 is out of bounds.
```

Let’s break this down.

The provided output includes disassembled code for different sections in an ELF file with the file format elf32-littlearm. Each section serves a specific purpose, and let's go through each one in detail:

Disassembly of **section .text**:
This section contains the machine code instructions of the program's executable code.

The Reset_Handler starts at address 0x00000000 and loads the value at address 0x10 into the r0 register. It then moves the value in r0 to the stack pointer (sp) and branches to the function c.
The Default_Handler starts at address 0x00000008 and contains a breakpoint instruction (bkpt) followed by an unconditional branch (b.n) to itself, creating an infinite loop.
The __start starts at address 0x0000000c and consists of a no-operation instruction (nop) followed by an unconditional branch to itself (b.n).

Disassembly of **section .isr_vector**:
This section contains the interrupt vector table, which holds the addresses of various interrupt service routines (ISRs). The actual content of this section is not shown in the provided output.

Disassembly of **section .debug_line**:
This section contains debugging information related to source code line numbers and mapping between source code and generated machine code.

Disassembly of **section .debug_info**:
This section contains debugging information about program entities such as variables, types, and functions.

Disassembly of **section .debug_abbrev**:
This section contains abbreviation tables used in debugging information to represent common data structures compactly.

Disassembly of **section .debug_aranges**:
This section contains address range information for debugging, helping map machine code addresses to source code line numbers during debugging sessions.

Disassembly of **section .debug_str**:
This section contains debug information strings, such as variable and function names, used during debugging.

Disassembly of **section .ARM.attributes**:
This section contains attributes specific to the ARM architecture, describing various characteristics of the binary.

Note: In the disassembly output, you may notice some lines with "Address X is out of bounds." This typically occurs when the disassembler encounters instructions that are invalid or when the disassembler is unable to determine the correct instruction due to data corruption or other issues in the ELF file.

It's important to remember that the disassembled output is a representation of the machine code instructions in human-readable form. It helps software developers understand the structure and behavior of the program, especially during debugging and analysis. The disassembled output alone may not provide the complete context of the program, as it may be linked with other object files and libraries to create the final executable or firmware for the STM32F401CCU6 microcontroller.

The next tool is a simplified version which displays assembler contents of just the executable sections.

```
arm-none-eabi-objdump -d main.o

main.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <Reset_Handler>:
   0:   4803            ldr     r0, [pc, #12]   ; (10 <__start+0x4>)
   2:   4685            mov     sp, r0
   4:   f000 f802       bl      c <__start>

00000008 <Default_Handler>:
   8:   be00            bkpt    0x0000
   a:   e7fd            b.n     8 <Default_Handler>

0000000c <__start>:
   c:   bf00            nop
   e:   e7fe            b.n     e <__start+0x2>
  10:   00000000        .word   0x00000000
```

The next tool displays the full contents of all sections requested.

```
arm-none-eabi-objdump -s main.o | less

main.o:     file format elf32-littlearm

Contents of section .text:
 0000 03488546 00f002f8 00befde7 00bffee7  .H.F............
 0010 00000000                             ....
Contents of section .isr_vector:
 0000 00000000 00000000 00000000 00000000  ................
 0010 00000000 00000000 00000000 00000000  ................
 0020 00000000 00000000 00000000 00000000  ................
 0030 00000000 00000000 00000000 00000000  ................
 0040 00000000 00000000 00000000 00000000  ................
 0050 00000000 00000000 00000000 00000000  ................
 0060 00000000 00000000 00000000 00000000  ................
 0070 00000000 00000000 00000000 00000000  ................
 0080 00000000 00000000 00000000 00000000  ................
 0090 00000000 00000000 00000000 00000000  ................
```

```
 00a0 00000000 00000000 00000000 00000000  ................
 00b0 00000000 00000000 00000000 00000000  ................
 00c0 00000000 00000000 00000000 00000000  ................
 00d0 00000000 00000000 00000000 00000000  ................
 00e0 00000000 00000000 00000000 00000000  ................
 00f0 00000000 00000000 00000000 00000000  ................
 0100 00000000 00000000 00000000 00000000  ................
 0110 00000000 00000000 00000000 00000000  ................
 0120 00000000 00000000 00000000 00000000  ................
 0130 00000000 00000000 00000000 00000000  ................
 0140 00000000 00000000 00000000 00000000  ................
 0150 00000000 00000000 00000000 00000000  ................
 0160 00000000 00000000 00000000 00000000  ................
 0170 00000000 00000000 00000000 00000000  ................
 0180 00000000 00000000 00000000 00000000  ................
 0190 00000000                             ....
Contents of section .debug_line:
 0000 40000000 03001d00 00000201 fb0e0d00  @...............
 0010 01010101 00000001 00000100 6d61696e  ............main
 0020 2e730000 00000000 05020000 000003b0  .s..............
 0030 01012121 030f2e21 030d2021 03602002  ..!!...!.. !.` .
 0040 02000101                             ....
Contents of section .debug_info:
 0000 22000000 02000000 00000401 00000000  "...............
 0010 00000000 14000000 00000000 07000000  ................
 0020 44000000 0180                        D.....
Contents of section .debug_abbrev:
 0000 01110010 06110112 01030e1b 0e250e13  .............%..
 0010 05000000                             ....
Contents of section .debug_aranges:
 0000 1c000000 02000000 00000400 00000000  ................
 0010 00000000 14000000 00000000 00000000  ................
Contents of section .debug_str:
 0000 6d61696e 2e730043 3a5c5573 6572735c  main.s.C:\Users\
 0010 6d797465 635c7374 6d333266 34303163  mytec\stm32f401c
 0020 6375362d 70726f6a 65637473 5c636c6f  cu6-projects\clo
 0030 7365745c 30783030 30312d74 656d706c  set\0x0001-templ
 0040 61746500 474e5520 41532032 2e33392e  ate.GNU AS 2.39.
 0050 353000                               50.
Contents of section .ARM.attributes:
 0000 41200000 00616561 62690001 16000000  A ...aeabi......
 0010 05436f72 7465782d 4d340006 0d074d09  .Cortex-M4....M.
 0020 02
```

The provided output displays the contents of different sections in an ELF file with the file format elf32-littlearm. Each section serves a specific purpose, and let's go through each one in detail:

Contents of **section .text**:
This section contains machine code instructions (binary code) for the main code of the program.

Contents of **section .isr_vector**:
This section contains the interrupt vector table, which holds the addresses of various interrupt service routines (ISRs). The actual content of this section is not shown in the provided output.

Contents of **section .debug_line**:
This section contains debugging information related to source code line numbers and mapping between source code and generated machine

code. It includes information about file names, directories, and line numbers.

Contents of **section .debug_info**:
This section contains debugging information about program entities such as variables, types, and functions. It includes detailed information to assist in debugging, symbol resolution, and source-level debugging.

Contents of **section .debug_abbrev**:
This section contains abbreviation tables used in debugging information to represent common data structures compactly. Abbreviations help reduce the size of debugging information.

Contents of **section .debug_aranges**:
This section contains address range information for debugging. It assists in mapping machine code addresses to source code line numbers during debugging sessions.

Contents of **section .debug_str**:
This section contains debug information strings, such as variable and function names, used during debugging. The strings provide human-readable names for the symbols in the binary.

Contents of **section .ARM.attributes**:
This section contains attributes specific to the ARM architecture, describing various characteristics of the binary. It includes information about the ARM architecture version, CPU features, and target architecture.

In summary, the output provides a glimpse of the contents stored in each section of the ELF file. These sections serve essential purposes during program execution and debugging, making it easier for developers to understand the behavior and structure of the program. Keep in mind that the disassembled output is shown in hexadecimal representation and may require additional tools or knowledge to interpret fully.

The next tool displays information about the contents of ELF format files.

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
```

```
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x8000194
  Start of program headers:          52 (bytes into file)
  Start of section headers:          69004 (bytes into file)
  Flags:                             0x5000200, Version5 EABI, soft-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         13
  Section header string table index: 12

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .isr_vector       PROGBITS        08000000 010000 000194 00   A  0   0  1
  [ 2] .text             PROGBITS        08000194 010194 000014 00  AX  0   0  4
  [ 3] .data             NOBITS          20000000 020000 000400 00  WA  0   0  1
  [ 4] .ARM.attributes   ARM_ATTRIBUTES  00000000 0101a8 000021 00      0   0  1
  [ 5] .debug_line       PROGBITS        00000000 0101c9 000044 00      0   0  1
  [ 6] .debug_info       PROGBITS        00000000 01020d 000026 00      0   0  1
  [ 7] .debug_abbrev     PROGBITS        00000000 010233 000014 00      0   0  1
  [ 8] .debug_aranges    PROGBITS        00000000 010248 000020 00      0   0  8
  [ 9] .debug_str        PROGBITS        00000000 010268 00004f 01  MS  0   0  1
  [10] .symtab           SYMTAB          00000000 0102b8 000520 10     11  15  4
  [11] .strtab           STRTAB          00000000 0107d8 000530 00      0   0  1
  [12] .shstrtab         STRTAB          00000000 010d08 000083 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), y (purecode), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x010000 0x08000000 0x08000000 0x001a8 0x001a8 R E 0x10000
  LOAD           0x000000 0x20000000 0x20000000 0x00000 0x00400 RW  0x10000

 Section to Segment mapping:
  Segment Sections...
   00     .isr_vector .text
   01     .data

There is no dynamic section in this file.

There are no relocations in this file.

There are no unwind sections in this file.

Symbol table '.symtab' contains 82 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08000000     0 SECTION LOCAL  DEFAULT    1 .isr_vector
     2: 08000194     0 SECTION LOCAL  DEFAULT    2 .text
     3: 20000000     0 SECTION LOCAL  DEFAULT    3 .data
     4: 00000000     0 SECTION LOCAL  DEFAULT    4 .ARM.attributes
     5: 00000000     0 SECTION LOCAL  DEFAULT    5 .debug_line
     6: 00000000     0 SECTION LOCAL  DEFAULT    6 .debug_info
     7: 00000000     0 SECTION LOCAL  DEFAULT    7 .debug_abbrev
     8: 00000000     0 SECTION LOCAL  DEFAULT    8 .debug_aranges
     9: 00000000     0 SECTION LOCAL  DEFAULT    9 .debug_str
    10: 00000000     0 FILE    LOCAL  DEFAULT  ABS main.o
    11: 08000194     0 NOTYPE  LOCAL  DEFAULT    2 $t
    12: 080001a1     0 FUNC    LOCAL  DEFAULT    2 __start
    13: 080001a4     0 NOTYPE  LOCAL  DEFAULT    2 $d
    14: 08000000     0 NOTYPE  LOCAL  DEFAULT    1 $d
    15: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI2_IRQHandler
```

```
    16: 0800019d     0 FUNC    WEAK   DEFAULT    2 DebugMon_Handler
    17: 0800019d     0 FUNC    WEAK   DEFAULT    2 SPI4_IRQHandler
    18: 0800019d     0 FUNC    WEAK   DEFAULT    2 TIM1_CC_IRQHandler
    19: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA2_Stream5_IRQ[...]
    20: 0800019d     0 FUNC    WEAK   DEFAULT    2 HardFault_Handler
    21: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA1_Stream5_IRQ[...]
    22: 0800019d     0 FUNC    WEAK   DEFAULT    2 SysTick_Handler
    23: 0800019d     0 FUNC    WEAK   DEFAULT    2 SDIO_IRQHandler
    24: 0800019d     0 FUNC    WEAK   DEFAULT    2 TAMP_STAMP_IRQHandler
    25: 0800019d     0 FUNC    WEAK   DEFAULT    2 PendSV_Handler
    26: 0800019d     0 FUNC    WEAK   DEFAULT    2 NMI_Handler
    27: 0800019d     0 FUNC    WEAK   DEFAULT    2 TIM1_BRK_TIM9_IR[...]
    28: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI3_IRQHandler
    29: 08000000     0 OBJECT  GLOBAL DEFAULT    1 isr_vector
    30: 0800019d     0 FUNC    WEAK   DEFAULT    2 TIM1_UP_TIM10_IR[...]
    31: 0800019d     0 FUNC    WEAK   DEFAULT    2 I2C3_ER_IRQHandler
    32: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI18_OTG_FS_WK[...]
    33: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI0_IRQHandler
    34: 0800019d     0 FUNC    WEAK   DEFAULT    2 I2C2_EV_IRQHandler
    35: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA1_Stream2_IRQ[...]
    36: 0800019d     0 FUNC    WEAK   DEFAULT    2 UsageFault_Handler
    37: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA2_Stream2_IRQ[...]
    38: 0800019d     0 FUNC    WEAK   DEFAULT    2 SPI1_IRQHandler
    39: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA2_Stream3_IRQ[...]
    40: 0800019d     0 FUNC    WEAK   DEFAULT    2 USART6_IRQHandler
    41: 08000195     0 FUNC    GLOBAL DEFAULT    2 Reset_Handler
    42: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA2_Stream0_IRQ[...]
    43: 0800019d     0 FUNC    WEAK   DEFAULT    2 TIM4_IRQHandler
    44: 0800019d     0 FUNC    WEAK   DEFAULT    2 I2C1_EV_IRQHandler
    45: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA1_Stream6_IRQ[...]
    46: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA1_Stream1_IRQ[...]
    47: 0800019d     0 FUNC    WEAK   DEFAULT    2 TIM3_IRQHandler
    48: 0800019d     0 FUNC    WEAK   DEFAULT    2 RCC_IRQHandler
    49: 0800019d     0 FUNC    GLOBAL DEFAULT    2 Default_Handler
    50: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI15_10_IRQHandler
    51: 0800019d     0 FUNC    WEAK   DEFAULT    2 ADC_IRQHandler
    52: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA1_Stream7_IRQ[...]
    53: 0800019d     0 FUNC    WEAK   DEFAULT    2 TIM5_IRQHandler
    54: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA2_Stream7_IRQ[...]
    55: 0800019d     0 FUNC    WEAK   DEFAULT    2 I2C3_EV_IRQHandler
    56: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI9_5_IRQHandler
    57: 0800019d     0 FUNC    WEAK   DEFAULT    2 SPI2_IRQHandler
    58: 0800019d     0 FUNC    WEAK   DEFAULT    2 MemManage_Handler
    59: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA1_Stream0_IRQ[...]
    60: 0800019d     0 FUNC    WEAK   DEFAULT    2 SVC_Handler
    61: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI4_IRQHandler
    62: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI22_RTC_WKUP_[...]
    63: 0800019d     0 FUNC    WEAK   DEFAULT    2 TIM2_IRQHandler
    64: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI16_PVD_IRQHandler
    65: 0800019d     0 FUNC    WEAK   DEFAULT    2 TIM1_TRG_COM_TIM[...]
    66: 20000400     0 NOTYPE  GLOBAL DEFAULT    3 _estack
    67: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI1_IRQHandler
    68: 0800019d     0 FUNC    WEAK   DEFAULT    2 EXTI17_RTC_Alarm[...]
    69: 0800019d     0 FUNC    WEAK   DEFAULT    2 USART2_IRQHandler
    70: 0800019d     0 FUNC    WEAK   DEFAULT    2 I2C2_ER_IRQHandler
    71: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA2_Stream1_IRQ[...]
    72: 0800019d     0 FUNC    WEAK   DEFAULT    2 FLASH_IRQHandler
    73: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA2_Stream4_IRQ[...]
    74: 0800019d     0 FUNC    WEAK   DEFAULT    2 BusFault_Handler
    75: 0800019d     0 FUNC    WEAK   DEFAULT    2 USART1_IRQHandler
    76: 0800019d     0 FUNC    WEAK   DEFAULT    2 OTG_FS_IRQHandler
    77: 0800019d     0 FUNC    WEAK   DEFAULT    2 SPI3_IRQHandler
    78: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA1_Stream4_IRQ[...]
    79: 0800019d     0 FUNC    WEAK   DEFAULT    2 I2C1_ER_IRQHandler
    80: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA2_Stream6_IRQ[...]
    81: 0800019d     0 FUNC    WEAK   DEFAULT    2 DMA1_Stream3_IRQ[...]

No version information found in this file.
Attribute Section: aeabi
File Attributes
```

```
  Tag_CPU_name: "Cortex-M4"
  Tag_CPU_arch: v7E-M
  Tag_CPU_arch_profile: Microcontroller
  Tag_THUMB_ISA_use: Thumb-2
```

Let’s break this down.

The provided output contains detailed information about an ELF (Executable and Linkable Format) file for an ARM architecture. Let's go through each part of the output and explain it in detail:

ELF Header:

Magic: This indicates the file type and that it is an ELF file.

Class: Specifies that it is an ELF32 (32-bit) file.

Data: Indicates it is stored in little-endian format (least significant byte first).

Version: The version of the ELF format (in this case, version 1, which is the current version).

OS/ABI: Specifies that it is targeting a UNIX System V-based operating system.

ABI Version: The version of the ABI (Application Binary Interface) used (in this case, version 0).

Type: Indicates that it is an EXEC (Executable file).

Machine: Specifies the target architecture (ARM).

Entry point address: The memory address where program execution begins (0x8000194 in this case).

Start of program headers: The offset in the file where the program headers start (52 bytes into the file).

Start of section headers: The offset in the file where the section headers start (69004 bytes into the file).

Flags: Additional information about the file (Version5 EABI, soft-float ABI).

Size of this header: The size of the ELF header in bytes (52 bytes).
Size of program headers: The size of a program header entry in bytes (32 bytes).

Number of program headers: The number of program header entries (2 in this case).

Size of section headers: The size of a section header entry in bytes (40 bytes).

Number of section headers: The number of section header entries (13 in this case).

Section header string table index: The index of the section header string table.

Section Headers:
This section provides information about each section in the ELF file, including the section's name, type, address, offset in the file, size, and other flags. The section headers store details about the various sections present in the file, such as code, data, debugging information, symbol tables, and more.

Program Headers:
This section describes the program segments and their corresponding attributes in the executable. Program headers are used to specify the segments that need to be loaded into memory during program execution. It includes information like segment type, virtual address, physical address, file size, memory size, and alignment.

Section to Segment mapping:
This table shows the mapping between sections and program segments. It indicates which sections are included in each program segment. Segments are used during the loading of the executable into memory.

Symbol table '.symtab':
This section contains entries for symbols present in the binary. Each entry includes information about the symbol's name, type, value (address), size, binding, visibility, and index.

Attribute Section: aeabi:
This section contains attribute information specific to the ARM architecture and follows the ARM EABI (Embedded Application Binary Interface) standard. It includes information about the CPU name, CPU architecture, CPU architecture profile, and the Thumb-2 ISA (Instruction Set Architecture) usage.

Overall, the output provides a comprehensive overview of the ELF file's structure and key information about sections, program headers, symbols, and attributes. This information is essential for the operating system and linking tools to load and execute the binary correctly.

Finally, we will look at a tool to list symbols in the file.

```
arm-none-eabi-nm main.elf | less

080001a0 t __start
20000400 B _estack
0800019c W ADC_IRQHandler
0800019c W BusFault_Handler
0800019c W DebugMon_Handler
0800019c T Default_Handler
0800019c W DMA1_Stream0_IRQHandler
0800019c W DMA1_Stream1_IRQHandler
0800019c W DMA1_Stream2_IRQHandler
0800019c W DMA1_Stream3_IRQHandler
0800019c W DMA1_Stream4_IRQHandler
0800019c W DMA1_Stream5_IRQHandler
0800019c W DMA1_Stream6_IRQHandler
0800019c W DMA1_Stream7_IRQHandler
0800019c W DMA2_Stream0_IRQHandler
0800019c W DMA2_Stream1_IRQHandler
0800019c W DMA2_Stream2_IRQHandler
0800019c W DMA2_Stream3_IRQHandler
0800019c W DMA2_Stream4_IRQHandler
0800019c W DMA2_Stream5_IRQHandler
0800019c W DMA2_Stream6_IRQHandler
0800019c W DMA2_Stream7_IRQHandler
0800019c W EXTI0_IRQHandler
0800019c W EXTI1_IRQHandler
0800019c W EXTI15_10_IRQHandler
0800019c W EXTI16_PVD_IRQHandler
0800019c W EXTI17_RTC_Alarm_IRQHandler
0800019c W EXTI18_OTG_FS_WKUP_IRQHandler
0800019c W EXTI2_IRQHandler
0800019c W EXTI22_RTC_WKUP_IRQHandler
0800019c W EXTI3_IRQHandler
0800019c W EXTI4_IRQHandler
0800019c W EXTI9_5_IRQHandler
0800019c W FLASH_IRQHandler
0800019c W HardFault_Handler
0800019c W I2C1_ER_IRQHandler
0800019c W I2C1_EV_IRQHandler
0800019c W I2C2_ER_IRQHandler
0800019c W I2C2_EV_IRQHandler
0800019c W I2C3_ER_IRQHandler
0800019c W I2C3_EV_IRQHandler
08000000 R isr_vector
0800019c W MemManage_Handler
0800019c W NMI_Handler
0800019c W OTG_FS_IRQHandler
0800019c W PendSV_Handler
0800019c W RCC_IRQHandler
08000194 T Reset_Handler
0800019c W SDIO_IRQHandler
0800019c W SPI1_IRQHandler
0800019c W SPI2_IRQHandler
0800019c W SPI3_IRQHandler
0800019c W SPI4_IRQHandler
0800019c W SVC_Handler
0800019c W SysTick_Handler
0800019c W TAMP_STAMP_IRQHandler
0800019c W TIM1_BRK_TIM9_IRQHandle
0800019c W TIM1_CC_IRQHandler
0800019c W TIM1_TRG_COM_TIM11_IRQHandler
0800019c W TIM1_UP_TIM10_IRQHandler
```

```
0800019c W TIM2_IRQHandler
0800019c W TIM3_IRQHandler
0800019c W TIM4_IRQHandler
0800019c W TIM5_IRQHandler
0800019c W UsageFault_Handler
0800019c W USART1_IRQHandler
0800019c W USART2_IRQHandler
0800019c W USART6_IRQHandler
```

Let’s break this down.

The provided output represents the symbol table ('.symtab') of an ELF file, which contains information about various symbols present in the binary. Symbols are identifiers used in the code, such as functions, variables, and other program elements. Each symbol has associated attributes, including its name, value (address), size, type, binding, and visibility. Let's go through the output and explain the symbols:

Symbols starting with '080001a0':

't __start': This is a local symbol ('t' stands for 'text') with the name '__start' located at address 0x080001a0. It is typically the entry point of the program.

Symbols starting with '20000400':

'B _estack': This is a global symbol ('B' stands for 'bss') with the name '_estack' located at address 0x20000400. It represents the bottom of the stack (end of memory) in RAM.

Symbols starting with '0800019c':

'W ADC_IRQHandler': This is a weak global symbol ('W' stands for 'weak') with the name 'ADC_IRQHandler' located at address 0x0800019c. It represents an interrupt service routine (ISR) for handling ADC interrupts.

'W BusFault_Handler': This is a weak global symbol representing the Bus Fault Handler ISR.

'W DebugMon_Handler': This is a weak global symbol representing the Debug Monitor Handler ISR.

'T Default_Handler': This is a global symbol ('T' stands for 'text') representing the Default Handler ISR.

'W DMA1_Stream0_IRQHandler' to 'W DMA2_Stream7_IRQHandler': These are weak global symbols representing different DMA Stream ISRs.
Symbol '08000000':

'R isr_vector': This is a global symbol ('R' stands for 'read-only data') representing the start of the interrupt vector table (usually called 'isr_vector') located at address 0x08000000.
Symbols starting with '08000194':

'T Reset_Handler': This is a global symbol ('T' stands for 'text') representing the Reset Handler ISR located at address 0x08000194.
Symbols starting with '0800019c':

'W SysTick_Handler': This is a weak global symbol representing the SysTick Handler ISR.

'W TAMP_STAMP_IRQHandler': This is a weak global symbol representing an ISR for handling Tamper and TimeStamp interrupts.
'W TIM1_BRK_TIM9_IRQHandle' to 'W TIM5_IRQHandler': These are weak global symbols representing different TIMx (Timer) ISRs.
Symbols starting with '0800019d':

'W EXTI0_IRQHandler' to 'W EXTI9_5_IRQHandler': These are weak global symbols representing different External Interrupt (EXTI) ISRs.
Symbols starting with '0800019c':

'W FLASH_IRQHandler': This is a weak global symbol representing the Flash memory interface ISR.

'W HardFault_Handler': This is a weak global symbol representing the Hard Fault Handler ISR.
Symbols starting with '0800019c':

'W I2C1_ER_IRQHandler' to 'W I2C3_EV_IRQHandler': These are weak global symbols representing different I2C ISRs.

Symbols starting with '0800019c':

'W NMI_Handler': This is a weak global symbol representing the Non-Maskable Interrupt (NMI) Handler ISR.

'W OTG_FS_IRQHandler': This is a weak global symbol representing the USB On-The-Go Full-Speed (OTG_FS) ISR.

Symbols starting with '0800019c':

'W PendSV_Handler': This is a weak global symbol representing the Pendable Service (PendSV) Handler ISR.

'W RCC_IRQHandler': This is a weak global symbol representing the Reset and Clock Control (RCC) ISR.

Symbols starting with '0800019c':

'W SDIO_IRQHandler': This is a weak global symbol representing the Secure Digital Input/Output (SDIO) ISR.

'W SPI1_IRQHandler' to 'W SPI4_IRQHandler': These are weak global symbols representing different SPI (Serial Peripheral Interface) ISRs.

Symbols starting with '0800019c':

'W SVC_Handler': This is a weak global symbol representing the Supervisor Call (SVC) Handler ISR.

'W TIM1_CC_IRQHandler': This is a weak global symbol representing the Timer 1 Capture/Compare (TIM1_CC) ISR.

Symbols starting with '0800019c':

'W UsageFault_Handler': This is a weak global symbol representing the Usage Fault Handler ISR.

'W USART1_IRQHandler' to 'W USART6_IRQHandler': These are weak global symbols representing different USART (Universal Synchronous/Asynchronous Receiver/Transmitter) ISRs.

These symbols represent the various interrupt service routines (ISRs) and handlers defined in the program. Each symbol's type and attributes are essential for the linker and debugger to correctly resolve and manage these symbols during the program's execution.

In our next chapter we will discuss the ARM Cortex-M registers.

Chapter 6: ARM Cortex-M Registers

Today we will begin our examination into the Cortex-M non-peripheral registers.

The Cortex-M has the following non-peripheral registers.

17 General-Purpose Registers
1 Status Register
3 Interrupt Mask Registers

Of the 17 GP registers, R0 to R12 are completely free to work with to hold variable 32-bit data value you want.

In ARM architecture, R13 is known as the Stack Pointer (SP). In the context of Cortex-M processors (which are part of the ARM architecture), it serves a crucial role in managing the stack. The stack is a region of memory used to store information during the execution of a program, especially during function calls and interrupt handling.

Here's how the Stack Pointer (R13) is used and its significance:

Stack Management: The Stack Pointer (SP) is a special register that points to the top of the stack, which is the last address used on the stack. The stack typically grows from higher memory addresses to lower memory addresses. When the stack is empty, the SP points to the highest address of the stack space.

Function Calls: Before a function is called, the caller saves its return address (the address of the instruction following the function call) in the Link Register (LR). Additionally, if the function being called has any local variables or needs to save certain register values across the function call (such as R4-R11), it allocates space on the stack.

Stack Frame: Each function call creates a new "stack frame" on the stack. A stack frame is a block of memory that holds the function's return address, saved registers, and local variables. It helps keep track of the function's execution context.

Nested Function Calls: When a function calls another function (nested function calls), each function gets its own stack frame, allowing multiple instances of the same function to be active simultaneously without interfering with each other's data.

Stack Management During Function Execution: As a function executes, it can push additional data onto the stack or pop data off the stack as needed. For example, when a function makes a local variable, it typically allocates space on the stack, and when that variable goes out of scope (function exits), that space is deallocated.

Stack Pointer Operations: The Stack Pointer (SP) is automatically adjusted by hardware during stack push (store) and pop (load) operations. For example, when a value is pushed onto the stack, the SP is decremented to point to the next available memory location for the next push operation. Similarly, when a value is popped from the stack, the SP is incremented to release that memory location for future use.

In summary, the Stack Pointer (R13) plays a critical role in managing the stack and maintaining the execution context of a program during function calls and interrupt handling. It points to the top of the stack and is automatically adjusted during push and pop operations to allocate and deallocate space for function call information and local variables.

In the ARM architecture, R14 is the Link Register (LR). The Link Register is a core register in the ARM Cortex-M architecture and is essential for managing function calls and returning from subroutines (functions).

The significance of the Link Register (R14) lies in its role during function calls and returning from function calls:

Function Calls: Before a function call is made, the calling function (caller) typically stores its return address in the Link Register (LR). The return address is the memory address of the instruction following the function call instruction. By storing this address in the LR, the processor knows where to return once the called function (callee) completes its execution.

Function Prologue: When a function is called, it sets up its stack frame, which includes saving the current LR value on the stack along with any other relevant register values (e.g., R4-R11) that need to be preserved across the function call. This allows the called function to have its own local variables and not interfere with the caller's variables.

Returning from Function Calls: When the called function completes its execution, it uses the LR value stored in the stack frame to return control to the calling function. The processor loads the LR value from the stack and jumps to the address stored in it, effectively resuming the execution of the calling function at the point just after the function call.

Efficient Subroutine Calls: The use of the LR as the return address allows for efficient subroutine calls, as it avoids the need to explicitly push the return address onto the stack before calling a function and then pop it back off afterward. This is particularly beneficial for embedded systems with limited resources like the STM32F401CCU6.

Nested Function Calls: In case of nested function calls (function A calls function B, which calls function C, and so on), the LR helps in maintaining the call chain. Each function call stores its return address in its respective LR, allowing for proper return flow when each function completes its execution.

Interrupt Handling: The LR is also crucial during interrupt handling. When an interrupt occurs, the processor automatically saves the current LR value onto the stack before jumping to the interrupt service routine (ISR). After the ISR completes its execution, it loads the LR value from the stack to return to the interrupted program flow.

In summary, the Link Register (R14) is a vital register in the STM32F401CCU6 microcontroller's ARM Cortex-M core. It facilitates function calls and returns, allowing for efficient subroutine calls and proper management of nested function calls and interrupt handling. Its use is critical for maintaining the execution flow and context in the system.

In the ARM archecture, R15 is the Program Counter (PC). The Program Counter is a core register in the ARM Cortex-M architecture, and its significance lies in its role in keeping track of the currently executing instruction and managing the program flow.

Here's the significance of the Program Counter (R15) in the STM32F401CCU6:

Instruction Fetch: The PC holds the memory address of the next instruction to be fetched and executed by the processor. During the fetch-execute cycle, the PC is used to fetch the instruction from memory, and after executing that instruction, the PC is automatically updated to point to the next instruction in memory.

Sequential Execution: As the name suggests, the Program Counter maintains the sequence of instruction execution. It ensures that the processor follows the correct order of instructions specified in the program, executing them one after the other.

Branch and Jump Instructions: When the processor encounters branch or jump instructions (e.g., B, BL, BX, BLX), the PC is modified to point to the target address specified by these instructions. This allows the processor to change the normal sequential flow of the program and jump to different parts of the code based on specific conditions or function calls.

Subroutine Calls and Returns: The PC plays a crucial role during subroutine calls and returns. When a subroutine is called (using BL or BLX), the PC is saved in the Link Register (LR), and the PC is then updated to the address of the subroutine. After the subroutine completes its execution, the PC is restored from the LR to resume execution at the instruction following the subroutine call (BL or BLX).

Interrupt Handling: During interrupt handling, the PC is automatically saved by the processor onto the stack when an interrupt occurs. This allows the processor to return to the interrupted program flow after handling the interrupt by restoring the PC from the stack.

Exception Handling: In addition to regular interrupts, the PC is also used in exception handling for various events such as faults, aborts, and system calls. The processor saves the PC onto the stack during exception entry and restores it during exception exit to resume normal program flow.

In summary, the Program Counter (R15) in the STM32F401CCU6 is fundamental for managing the program flow and instruction execution. It keeps track of the next instruction to be executed, allows for branching and jumping to different parts of the code, facilitates subroutine calls and returns, and plays a vital role during interrupt and exception handling. The correct operation of the Program Counter is essential for the proper execution of the program and overall system functionality.

In the ARM Cortex-M4 processor, including the STM32F401CCU6, the Program Status Register (PSR) contains important status and control information about the processor's current execution state. The PSR consists of three main fields:

APSR (Application Program Status Register): This field contains various application program status flags, including the zero flag (Z), the negative flag (N), the carry flag (C), the overflow flag (V), and the Q flag (for saturation arithmetic).

IPSR (Interrupt Program Status Register): This field indicates the exception number of the currently executing exception (interrupt or fault). In thumb mode, the IPSR is part of the PSR (XPSR) and indicates the current active exception number.

EPSR (Execution Program Status Register): This field holds execution status flags such as the Thumb bit (T), the stack pointer selection bit (SPSEL), and the floating-point extension enable bit (FPCA).

For the specific STM32F401CCU6, we'll focus on the APSR and some relevant bits in the EPSR:

APSR (Application Program Status Register):

N (Negative) Flag (Bit 31): Set when the result of an operation is negative. For example, after a subtraction where the result is less than zero.

Z (Zero) Flag (Bit 30): Set when the result of an operation is zero. For example, after a subtraction where the result is equal to zero.

C (Carry) Flag (Bit 29): Set when there is a carry or borrow out of the most significant bit in arithmetic and logical operations, such as addition or subtraction.

V (Overflow) Flag (Bit 28): Set when a signed arithmetic operation results in overflow or underflow, indicating that the result does not fit within the available bits.

Q (Saturation) Flag (Bit 27): Set when saturation arithmetic is enabled (optional in ARM Cortex-M4) and an arithmetic operation results in saturation.

EPSR (Execution Program Status Register):

T (Thumb Bit) (Bit 24): Set when the processor is in Thumb state. In the STM32F401CCU6, the processor operates mainly in Thumb state, which allows for more compact code and better power efficiency compared to ARM state.

SPSEL (Stack Pointer Select) Bit (Bit 1): Set to 0 when the main stack pointer (MSP) is selected and set to 1 when the process stack pointer (PSP) is selected. The processor can switch between these two stack pointers for different execution contexts.

FPCA (Floating-Point Context Active) Bit (Bit 2): Set to 0 when the Floating-Point Unit (FPU) context is not active and set to 1 when an FPU context is active. This bit indicates whether floating-point instructions can be executed.

These status flags are critical for conditional branching and controlling the flow of the program based on the results of arithmetic and logical operations. They also play a role in exception handling and debugging by providing information about the processor's current state.

Keep in mind that some features, such as the FPU and saturation arithmetic, are optional in the Cortex-M4 architecture and may not be present in all implementations, including the STM32F401CCU6. The specific implementation details can be found in the device's technical reference manual or datasheet.

In ARM assembly language, MRS stands for "Move from Special Register." It is an instruction used to read the value of a special register in the ARM Cortex-M processor. Special registers in the Cortex-M architecture are typically system control registers, status registers, or configuration registers that control various aspects of the processor's behavior or provide information about its current state.

The MRS instruction allows you to transfer the contents of a special register to a general-purpose register, where you can then perform further operations or use the value as needed in your program. The general syntax of the MRS instruction is as follows:

```
MRS <Rd>, <special_register>
```

Here, <Rd> is the destination general-purpose register where the value of the special register will be stored, and <special_register> represents the name of the special register you want to read.

For example, let's say you want to read the value of the APSR (Application Program Status Register) into R0. The instruction would look like this:

```
MRS R0, APSR
```

Similarly, you can read other special registers like IPSR (Interrupt Program Status Register), xPSR (Combined Program Status Register), MSP (Main Stack Pointer), PSP (Process Stack Pointer), and more.

It's important to note that accessing certain special registers might require privileged execution mode, and some registers may not be directly accessible in unprivileged mode. If the instruction is executed in unprivileged mode and the special register is not accessible, the behavior of the MRS instruction might result in an undefined operation or raise an exception.

To use special registers effectively, especially those related to system control and configuration, it's crucial to refer to the processor's reference manual or technical documentation to understand their purpose, accessibility, and potential side effects. Additionally, the availability and names of specific special registers may vary depending on the specific ARM Cortex-M processor variant being used.

In ARM assembly language, MSR stands for "Move to Special Register." It is an instruction used to write a value into a special register in the ARM Cortex-M processor. Special registers in the Cortex-M architecture are typically system control registers, status registers, or configuration registers that control various aspects of the processor's behavior or provide information about its current state.

The MSR instruction allows you to set the value of a special register using a value from a general-purpose register or an immediate value. The general syntax of the MSR instruction is as follows:

```
MSR <special_register>, <Rn>
```

Here, <special_register> represents the name of the special register you want to write to, and <Rn> is the source general-purpose register containing the value you want to write into the special register.

Alternatively, you can use an immediate value as the source to directly set the value of the special register. In this case, the syntax would be:

```
MSR <special_register>, #<immediate_value>
```

Here, <special_register> represents the name of the special register you want to write to, and <immediate_value> is the immediate value you want to set in the special register.

For example, if you want to set the value of the CONTROL register with the value in R0, the instruction would look like this:

```
MSR CONTROL, R0
```

Similarly, you can directly set the value of some special registers using an immediate value. For example, to set the PRIMASK (Priority Mask) register to 1, you can use:

```
MSR PRIMASK, #1
```

It's important to note that accessing certain special registers might require privileged execution mode, and some registers may not be directly writable in unprivileged mode. If the instruction is executed in unprivileged mode and the special register is not writable, the behavior of the MSR instruction might result in an undefined operation or raise an exception.

To use MSR effectively and safely, especially for system control and configuration, it's crucial to refer to the processor's reference manual or technical documentation to understand their purpose, accessibility, and potential side effects. Additionally, the availability and names of specific special registers may vary depending on the specific ARM Cortex-M processor variant being used.

In our next chapter we will discuss the ARM Thumb2 instruction set.

Chapter 7: ARM Thumb2 Instruction Set

Today we will begin our examination into the ARM Thumb2 instruction set.

The ARM Cortex-M4 processor used in the STM32F401CCU6 microcontroller implements the ARMv7-M architecture, which includes the Thumb-2 instruction set. Thumb-2 is a compact 16-bit and 32-bit mixed instruction set that combines the benefits of both the 16-bit Thumb instructions and the 32-bit ARM instructions. It allows for more code density and improved performance compared to the older Thumb and ARM instruction sets.

The Thumb-2 instruction set includes various types of instructions, and I'll explain some of the key categories and examples below:

Data Processing Instructions:

Add, Subtract, Multiply, and other arithmetic operations:

ADD Rd, Rn, Operand2: Adds the value in Rn to Operand2 and stores the result in Rd.

SUB Rd, Rn, Operand2: Subtracts the value in Operand2 from Rn and stores the result in Rd.

MUL Rd, Rn, Rm: Multiplies the values in Rn and Rm and stores the result in Rd.

Load and Store Instructions:

Load a value from memory into a register:
LDR Rd, [Rn, Offset]: Loads the value from memory at address Rn + Offset into Rd.

Store a value from a register into memory:
STR Rd, [Rn, Offset]: Stores the value from Rd into memory at address Rn + Offset.

Branch Instructions:

Unconditional branch:
B Label: Jumps to the instruction at Label.

Conditional branch:
BEQ/BNE/BGT/BLT, etc.: Branches to the Label if the specified condition is met.

Control Flow Instructions:

Subroutine Call:
BL Label: Calls the subroutine at Label and saves the return address in the link register (LR).

Return from Subroutine:
BX LR: Branches to the address stored in the link register, effectively returning from a subroutine.

Bit Manipulation Instructions:

Set and Clear individual bits:
BSET/BCLR: Sets or clears a specific bit in a register.

Shift and Rotate Instructions:

Shift or rotate the bits in a register:
LSL/LSR/ASR/ROR: Logical Shift Left/Right, Arithmetic Shift Right, Rotate Right.

Stack Instructions:

Push and Pop values from the stack:

PUSH: Pushes multiple registers onto the stack.

POP: Pops multiple registers from the stack.

These are just some examples of the Thumb-2 instructions available in the Cortex-M4 architecture. The Thumb-2 instruction set is designed to be efficient, enabling a good balance between code size and performance, which is especially crucial in microcontroller applications with limited resources.

The STM32F401CCU6 microcontroller's reference manual and Cortex-M4 Technical Reference Manual provide comprehensive information on the Thumb-2 instruction set and other architecture-specific details.

Directives are instructions used in assembly language programming to provide additional information to the assembler or linker. They don't represent machine instructions executed by the CPU; instead, they control how the assembler generates the machine code or how the linker organizes the final executable code. These directives are specific to the assembler being used and may vary between different architectures and toolchains. Below are explanations of some commonly used directives:

.space:
Syntax: .space size
Description: Reserves a block of memory of the specified size (in bytes) without initializing it. The memory is typically filled with zeros or left uninitialized, depending on the assembler and target architecture. This directive is useful for reserving space for variables or buffers.

.word:
Syntax: .word value1, value2, ...
Description: Initializes memory with a sequence of 32-bit (4-byte) values. Each value listed after the .word directive is stored consecutively in memory. For example, .word 10, 20, 30 would store the values 10, 20, and 30 in consecutive memory locations.

.section:
Syntax: .section section_name [, "flags"]
Description: Specifies a section or segment for the following code or data. A section is a logical unit used for grouping related code or data together. The optional "flags" argument can be used to provide additional information about the section, such as its permissions, alignment, etc.

.global or .globl:
Syntax: .global symbol or .globl symbol
Description: Declares a symbol as global, meaning it can be accessed from other source files or object files. This is necessary when you want to use a symbol defined in one source file in another source file.

.equ:
Syntax: .equ symbol, expression
Description: Defines a symbol with a constant value. The value of the symbol is computed based on the provided expression. For example, .equ my_constant, 42 would define the symbol my_constant with the value 42.

.align:
Syntax: .align alignment

Description: Adjusts the alignment of the following code or data to the specified value. The alignment value should be a power of 2, and the assembler inserts padding bytes, if necessary, to ensure that the next address is aligned correctly.

.text, .data, .bss, .rodata, etc.:
These are section names used to specify the type of data or code that follows. For example, .text is used for executable code, .data for initialized data, .bss for uninitialized data, and .rodata for read-only data.

It's important to note that the specific directives and their syntax may vary depending on the assembler and target architecture being used. The examples provided above are generic and may not represent the exact syntax used in a specific assembly language or toolchain. Therefore, it's essential to refer to the documentation of the assembler and the specific target architecture for accurate and up-to-date information.

In our next chapter we will discuss load & store instructions.

Chapter 8: Load & Store Instructions

Now the fun begins as we get to dive back into coding!

Let’s get our project setup below and copy over our template.

```
cd stm32f401ccu6-projects
mkdir 0x0002-load_and_store_instructions
cd 0x0002-load_and_store_instructions
cp ..\0x0001-template\main.s .
cp ..\0x0001-template\stm32f401ccux.ld .
```

In ARM assembly language, the LDR (Load Register) and STR (Store Register) instructions are used to load data from memory into a register and store data from a register into memory, respectively. These instructions are fundamental for accessing data in memory and are essential for various tasks in programming, such as reading and writing variables, arrays, and structures.

LDR (Load Register):
Syntax: LDR Rd, [Rn, #Offset]
Description: The LDR instruction loads a 32-bit word from memory into a register. The address to load from is computed as the sum of the base register Rn and the immediate offset Offset. The result is stored in the destination register Rd.

Example:

```
LDR R1, [R0, #4]
```

This instruction loads a 32-bit word from the memory address stored in R0 + 4 bytes into register R1.

Note: On the Cortex-M4, the Offset must be a multiple of 4, as it deals with 32-bit words.

STR (Store Register):
Syntax: STR Rd, [Rn, #Offset]
Description: The STR instruction stores the contents of a register into memory. The address to store into is computed as the sum of the base register Rn and the immediate offset Offset. The data in the source register Rd is stored in memory.

Example:

```
STR R1, [R0, #8]
```

This instruction stores the contents of register R1 into the memory address stored in R0 + 8 bytes.

Note: On the Cortex-M4, the Offset must be a multiple of 4, as it deals with 32-bit words.

LDR and STR with Immediate Offset:
Both LDR and STR instructions can use an immediate offset (positive or negative) to access memory locations relative to the base register.

Example:

```
LDR R2, [R3, #12]
```

This instruction loads a 32-bit word from the memory address stored in R3 + 12 bytes into register R2.

```
STR R4, [R5, #-16]
```

This instruction stores the contents of register R4 into the memory address stored in R5 - 16 bytes.

LDR and STR with Register Offset:
The LDR and STR instructions can also use a register as an offset to access memory locations.

Example:

```
LDR R6, [R7, R8]
```

This instruction loads a 32-bit word from the memory address stored in R7 + the value stored in R8 into register R6.

```
STR R9, [R10, -R11]
```

This instruction stores the contents of register R9 into the memory address stored in R10 - the value stored in R11.

These instructions are essential for data manipulation and memory access in ARM assembly language programming. It's important to ensure that memory addresses are correctly calculated and aligned, especially on the Cortex-M4 architecture, which requires 32-bit word alignment. Also, pay attention to the source and destination registers to avoid overwriting critical data during memory operations.

Let’s edit **main.s** and if you are unfamiliar with VIM please watch this video. https://youtu.be/ggSyF1SVFr4

```
/**
 * FILE: main.s
 *
 * DESCRIPTION:
 * This file contains the assembly code for a simple load and store firmware
 * example utilizing the STM32F401CC6 microcontroller.
 *
 * AUTHOR: Kevin Thomas
 * CREATION DATE: July 21, 2023
 * UPDATE Date: July 21, 2023
 *
 * ASSEMBLE AND LINK w/ SYMBOLS:
 * 1. arm-none-eabi-as -g main.s -o main.o
 * 2. arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
 * 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf verify reset exit"
 * ASSEMBLE AND LINK w/o SYMBOLS:
 * 1. arm-none-eabi-as -g main.s -o main.o
 * 2. arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
 * 3. arm-none-eabi-objcopy -O binary --strip-all main.elf main.bin
 * 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.bin 0x08000000 verify reset exit"
 * DEBUG w/ SYMBOLS:
 * 1. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
 * 2. arm-none-eabi-gdb main.elf
 * 3. target remote :3333
 * 4. monitor reset halt
 * 5. l
 * DEBUG w/o SYMBOLS:
 * 1. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
 * 2. arm-none-eabi-gdb main.bin
 * 3. target remote :3333
 * 4. monitor reset halt
 * 5. x/8i $pc
 */

.syntax unified
.cpu cortex-m4
.thumb

/**
 * Provide weak aliases for each Exception handler to the Default_Handler.
 * As they are weak aliases, any function with the same name will override
 * this definition.
 */
.macro weak name
  .global \name
  .weak \name
  .thumb_set \name, Default_Handler
  .word \name
.endm

/**
 * The STM32F401CCUx vector table.  Note that the proper constructs
 * must be placed on this to ensure that it ends up at physical address
 * 0x0000.0000.
 */
.global isr_vector
.section .isr_vector, "a"
.type isr_vector, %object
isr_vector:
  .word _estack
  .word Reset_Handler
   weak NMI_Handler
   weak HardFault_Handler
   weak MemManage_Handler
   weak BusFault_Handler
   weak UsageFault_Handler
  .word 0
  .word 0
  .word 0
  .word 0
   weak SVC_Handler
   weak DebugMon_Handler
  .word 0
   weak PendSV_Handler
   weak SysTick_Handler
  .word 0
   weak EXTI16_PVD_IRQHandler                           // EXTI Line 16 interrupt /PVD through EXTI line detection interrupt
   weak TAMP_STAMP_IRQHandler                           // Tamper and TimeStamp interrupts through the EXTI line
   weak EXTI22_RTC_WKUP_IRQHandler                      // EXTI Line 22 interrupt /RTC Wakeup interrupt through the EXTI line
   weak FLASH_IRQHandler                                // FLASH global interrupt
   weak RCC_IRQHandler                                  // RCC global interrupt
   weak EXTI0_IRQHandler                                // EXTI Line0 interrupt
   weak EXTI1_IRQHandler                                // EXTI Line1 interrupt
   weak EXTI2_IRQHandler                                // EXTI Line2 interrupt
   weak EXTI3_IRQHandler                                // EXTI Line3 interrupt
   weak EXTI4_IRQHandler                                // EXTI Line4 interrupt
   weak DMA1_Stream0_IRQHandler                         // DMA1 Stream0 global interrupt
   weak DMA1_Stream1_IRQHandler                         // DMA1 Stream1 global interrupt
   weak DMA1_Stream2_IRQHandler                         // DMA1 Stream2 global interrupt
   weak DMA1_Stream3_IRQHandler                         // DMA1 Stream3 global interrupt
   weak DMA1_Stream4_IRQHandler                         // DMA1 Stream4 global interrupt
   weak DMA1_Stream5_IRQHandler                         // DMA1 Stream5 global interrupt
   weak DMA1_Stream6_IRQHandler                         // DMA1 Stream6 global interrupt
```

```
   weak ADC_IRQHandler                                        // ADC1 global interrupt
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
   weak EXTI9_5_IRQHandler                                    // EXTI Line[9:5] interrupts
   weak TIM1_BRK_TIM9_IRQHandle                               // TIM1 Break interrupt and TIM9 global interrupt
   weak TIM1_UP_TIM10_IRQHandler                              // TIM1 Update interrupt and TIM10 global interrupt
   weak TIM1_TRG_COM_TIM11_IRQHandler                         // TIM1 Trigger and Commutation interrupts and TIM11 global interrupt
   weak TIM1_CC_IRQHandler                                    // TIM1 Capture Compare interrupt
   weak TIM2_IRQHandler                                       // TIM2 global interrupt
   weak TIM3_IRQHandler                                       // TIM3 global interrupt
   weak TIM4_IRQHandler                                       // TIM4 global interrupt
   weak I2C1_EV_IRQHandler                                    // I2C1 event interrupt
   weak I2C1_ER_IRQHandler                                    // I2C1 error interrupt
   weak I2C2_EV_IRQHandler                                    // I2C2 event interrupt
   weak I2C2_ER_IRQHandler                                    // I2C2 error interrupt
   weak SPI1_IRQHandler                                       // SPI1 global interrupt
   weak SPI2_IRQHandler                                       // SPI2 global interrupt
   weak USART1_IRQHandler                                     // USART1 global interrupt
   weak USART2_IRQHandler                                     // USART2 global interrupt
  .word 0                                                     // Reserved
   weak EXTI15_10_IRQHandler                                  // EXTI Line[15:10] interrupts
   weak EXTI17_RTC_Alarm_IRQHandler                           // EXTI Line 17 interrupt / RTC Alarms (A and B) through EXTI line interrupt
   weak EXTI18_OTG_FS_WKUP_IRQHandler                         // EXTI Line 18 interrupt / USBUSB OTG FS Wakeup through EXTI line interrupt
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
   weak DMA1_Stream7_IRQHandler                               // DMA1 Stream7 global interrupt
  .word 0                                                     // Reserved
   weak SDIO_IRQHandler                                       // SDIO global interrupt
   weak TIM5_IRQHandler                                       // TIM5 global interrupt
   weak SPI3_IRQHandler                                       // SPI3 global interrupt
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
   weak DMA2_Stream0_IRQHandler                               // DMA2 Stream0 global interrupt
   weak DMA2_Stream1_IRQHandler                               // DMA2 Stream1 global interrupt
   weak DMA2_Stream2_IRQHandler                               // DMA2 Stream2 global interrupt
   weak DMA2_Stream3_IRQHandler                               // DMA2 Stream3 global interrupt
   weak DMA2_Stream4_IRQHandler                               // DMA2 Stream4 global interrupt
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
   weak OTG_FS_IRQHandler                                     // USB On The Go FS global interrupt
   weak DMA2_Stream5_IRQHandler                               // DMA2 Stream5 global interrupt
   weak DMA2_Stream6_IRQHandler                               // DMA2 Stream6 global interrupt
   weak DMA2_Stream7_IRQHandler                               // DMA2 Stream7 global interrupt
   weak USART6_IRQHandler                                     // USART6 global interrupt
   weak I2C3_EV_IRQHandler                                    // I2C3 event interrupt
   weak I2C3_ER_IRQHandler                                    // I2C3 error interrupt
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
  .word 0                                                     // Reserved
   weak SPI4_IRQHandler                                       // SPI4 global interrupt

.section .text

/**
 * @brief  This code is called when processor starts execution.
 *
 *         This is the code that gets called when the processor first
 *         starts execution following a reset event. Only the absolutely
 *         necessary set is performed, after which the application
 *         supplied main() routine is called.
 * @param  None
 * @retval None
 */
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
  LDR   R0, =_estack                                          // load address at end of the stack into R0
  MOV   SP, R0                                                // move address at end of stack into SP
  BL    __start                                               // call function

/**
 * @brief  This code is called when the processor receives and unexpected interrupt.
 *
 *         This is the code that gets called when the processor receives an
 *         unexpected interrupt.  This simply enters an infinite loop, preserving
 *         the system state for examination by a debugger.
```

```
 *
 * @param  None
 * @retval None
 */
.type Default_Handler, %function
.global Default_Handler
Default_Handler:
  BKPT                                    // set processor into debug state
  B.N   Default_Handler                   // call function, force thumb state

/**
 * @brief  Entry point for initialization and setup of specific functions.
 *
 *         This function is the entry point for initializing and setting up specific functions.
 *         It calls other functions to enable certain features and then enters a loop for further execution.
 *
 * @param  None
 * @retval None
 */
.type __start, %function
__start:
  LDR  R0, =0x40023830                    // load address of RCC_AHB1ENR register
  LDR  R1, [R0]                           // load value inside RCC_AHB1ENR register
  ORR  R1, #(1<<0)                        // set the GPIOAEN bit
  STR  R1, [R0]                           // store value into RCC_AHB1ENR register
  B    .                                  // branch infinite loop
```

The above contains the entire code of the firmware. What we are most interested in is what gets executed in the __start function.

Let's explain what is going on by first assembling then linking and finally flashing to our MCU.

```
arm-none-eabi-as -g main.s -o main.o
```

```
arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
```

```
openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf verify reset exit"
```

Now let's fire up our debugger to peek inside!

Terminal 1:

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

Terminal 2:

```
arm-none-eabi-gdb main.elf
```

```
target remote :3333
```

```
monitor reset halt
```

Now that we are inside the firmware, lets set a breakpoint on our __start function which is what is directly executed after the Reset_Handler. Let's continue and disassemble.

```
(gdb) b __start
Breakpoint 1 at 0x80001a0: file main.s, line 208.
Note: automatically using hardware breakpoints for read-only addresses.
(gdb) c
Continuing.


Breakpoint 1, __start () at main.s:208
208       LDR  R0, =0x40023830                    // load address of RCC_AHB1ENR register
(gdb) disas
```

```
Dump of assembler code for function __start:
=> 0x080001a0 <+0>:     ldr     r0, [pc, #12]   ; (0x80001b0 <__start+16>)
   0x080001a2 <+2>:     ldr     r1, [r0, #0]
   0x080001a4 <+4>:     orr.w   r1, r1, #1
   0x080001a8 <+8>:     str     r1, [r0, #0]
   0x080001aa <+10>:    b.n     0x80001aa <__start+10>
   0x080001ac <+12>:    lsls    r0, r0, #16
   0x080001ae <+14>:    movs    r0, #0
   0x080001b0 <+16>:    subs    r0, #48 ; 0x30
   0x080001b2 <+18>:    ands    r2, r0
End of assembler dump.
```

The first thing we notice is our code is currently about to execute 0x080001a0 as you see the => before the address.

The ARM assembly instruction ldr r0, [pc, #12] is used to load a word (32-bit value) from memory into register r0. Let's break down the instruction step by step:

ldr: This is the mnemonic for the Load (LDR) instruction. It is used to load a value from memory into a register.

r0: This is the destination register where the value will be loaded. In this case, the value from memory will be loaded into register r0.

[pc, #12]: This is the memory address from where the value will be loaded. The square brackets [] indicate that it's an indirect memory access. pc stands for the Program Counter, which points to the current instruction address. #12 is an immediate offset value, meaning it is a constant value that is added to the pc to calculate the memory address.

To understand how this instruction works, you need to consider the addressing mode and the memory layout of the ARM processor.

Addressing Mode:
In this instruction, the addressing mode used is [pc, #12], which is known as PC-relative addressing mode. It allows you to access data in memory relative to the current instruction address (PC).

Memory Layout (Little-Endian):
In ARM processors, data is stored in memory in little-endian format. This means that the least significant byte of a word is stored at the lower memory address, and the most significant byte is stored at the higher memory address.

Explanation of the Instruction:

The instruction ldr r0, [pc, #12] is executed.

The value of the Program Counter (PC) is determined, which points to the address of the current instruction.

An offset of 12 bytes is added to the PC to calculate the memory address from which the word will be loaded.

The word value (32 bits) located at the calculated memory address is loaded into register r0.

Assuming that the current instruction's address (PC) is 0x08001234, the memory address accessed by this instruction would be 0x08001234 + 12 = 0x08001240.

For example, if the memory at address 0x08001240 contains the value 0xABCD1234, then the ldr r0, [pc, #12] instruction will load 0xABCD1234 into register r0.

Note: The actual memory address accessed by the instruction depends on the current PC and the value of the offset (#12). The offset value can be positive or negative, depending on the instruction's location relative to the data being accessed.

Let’s first see what value is inside R0.

```
(gdb) i r r0
r0             0x20000400          536871936
```

We see 0x20000400 in hex. We need to remember that in the Reset_Handler, we in fact move the value of _estack into R0 so that is where this value is coming from.

Let’s si once and see what happens to R0 once we execute the LDR instruction.

```
(gdb) si
209       LDR  R1, [R0]                                            // load value inside RCC_AHB1ENR register
(gdb) i r r0
r0             0x40023830          1073887280
```

We see that R0 now holds the address of 0x40023830.

If we do another disassembly we can see that our PC moved up to the next instruction and we will see a new line => being pointed to.

```
(gdb) disas
Dump of assembler code for function __start:
   0x080001a0 <+0>:     ldr     r0, [pc, #12]   ; (0x80001b0 <__start+16>)
=> 0x080001a2 <+2>:     ldr     r1, [r0, #0]
   0x080001a4 <+4>:     orr.w   r1, r1, #1
   0x080001a8 <+8>:     str     r1, [r0, #0]
   0x080001aa <+10>:    b.n     0x80001aa <__start+10>
   0x080001ac <+12>:    lsls    r0, r0, #16
   0x080001ae <+14>:    movs    r0, #0
   0x080001b0 <+16>:    subs    r0, #48 ; 0x30
   0x080001b2 <+18>:    ands    r2, r0
End of assembler dump.
```

We are about to execute the next instruction. We see that whatever is inside R0 with an offset of 0 will be placed into R1.
Keep in mind, we know that R0 holds a memory address however when we use [] this will take the value inside the memory address and then store that into R1. Let’s examine!

```
(gdb) si
210       ORR  R1, #(1<<0)                                      // set the GPIOAEN bit
(gdb) i r r1
r1             0x0                 0
```

So it is clear that the initial value inside the memory address of 0x40023830 is 0x00.

The very next instruction is the ORR instruction to which we are going to set the 0 bit, as there are 32 total bits in this register starting from 0 and ending on 31, to 1.

Currently we know that the value is 0x00 in hex or 0x00000000000000000000000000000000 in binary.

We use ORR to set that 0^{th} bit to 1 without disturbing any other bit status as our debug shows orr.w r1, r1, #1 which is the same thing as our code which is ORR R1, #(1<<0). Lets take a moment and understand what is going on here.

Let's break down the ARM assembly instruction ORR R1, #(1<<0) step by step:

ORR: This is the mnemonic for the ORR (OR with immediate) instruction. It performs a bitwise OR operation between the contents of a register and an immediate value (constant) and stores the result in the destination register.

R1: This is the destination register. The result of the OR operation will be stored in register R1.

#(1<<0): This is the immediate value being used as the second operand in the ORR instruction. (1<<0) means 1 is left-shifted 0 bits, which essentially means the immediate value is 1. In other words, it's the binary number 00000001.

Now let's understand the operation:

The ORR instruction performs a bitwise OR operation between the contents of register R1 (the destination register) and the immediate value 1.

The bitwise OR operation takes two binary numbers and produces a result where each bit in the result is 1 if at least one of the corresponding bits in the two input numbers is 1. Otherwise, the bit in the result is 0.

Let's consider the binary representation of the initial value in register R1 (before the OR operation) and the immediate value 1:

```
R1:  00000000 00000000 00000000 00000000
1:   00000000 00000000 00000000 00000001
```

Performing the bitwise OR operation:

```
Result: 00000000 00000000 00000000 00000001
```

The result of the OR operation is 1 (binary 00000001).

Finally, the value 1 is stored back in register R1.

So, after executing ORR R1, #(1<<0), register R1 will contain the value 1.

This operation is commonly used in embedded systems programming to set specific bits in a register or a memory-mapped hardware control register. By using the ORR instruction with specific immediate values, you can set individual bits in a register to enable or disable specific functionalities or configurations.

Let's disassemble and prove this.

```
(gdb) si
211       STR  R1, [R0]                                         // store value into RCC_AHB1ENR register
(gdb) i r r1
r1             0x1                 1
```

Now we are going to take our value in R1 which is 0x01 and store that into the value that is stored in R0.

```
(gdb) si
212       B    .                                                // branch infinite loop
(gdb) i r r0
r0             0x40023830          1073887280
(gdb) x/x $r0
0x40023830:     0x00000001
```

The x/x $r0 means, tell me the value inside the address which R0 points to and in this case it is 0x01.

Now we know that 0x40023830 is one of our peripheral addresses that communicates over the system bus. We an also examine that the value at this address has in fact been changed.

```
(gdb) x/x 0x40023830
0x40023830:     0x00000001
```

As you are hopefully starting to see is that you have ABSOLUTE domain over this MCU as there is NOTHING that we are not covering or not understanding.

Software abstractions are necessary in rapid development however are a cancer for TRULY understanding what is ACTUALLY going on under the hood and therefore the reason why I spend years writing free books to help educate and teach the realities of how things work especially when we are under attack from every thing cyber!

Getting off my soapbox, we can also literally set values within the peripheral registers directly!

Imagine we have a debug session into a foreign IoT device and this address controls the GPIOA access to the clock such that if we disable it, it will render all GPIOA instructions useless and will NOT cause an error!

IMAGINE THE POWER YOU HAVE WITH THIS KNOWLEDGE! You could disable a warning light, LED or anything for that matter.

Lets prove this!

We know 0x40023830 currently has the value 0x01.

```
(gdb) x/x 0x40023830
0x40023830:     0x00000001
```

Let’s hack this live!

```
(gdb) set *(0x40023830) = 0x00
```

Now the moment of truth!

```
(gdb) x/x 0x40023830
0x40023830:     0x00000000
```

WOOHOO! We did it! In addition no one would be the wiser!

At this point I would highly encourage you to research Stuxnet if you have not already ;)

These skills that you are learning will help protect and manipulate IoT devices in the wild and this skill is ESSENTIAL to our survival!

In our next lesson we will learn about constants and literal values.

Chapter 9: Constants & Literal Values

Let’s talk about constants and literals.

Let’s get our project setup below and copy over our template.

```
cd stm32f401ccu6-projects
mkdir 0x0003-constants_and_literal_values
cd 0x0003-constants_and_literal_values
cp ..\0x0001-template\main.s .
cp ..\0x0001-template\stm32f401ccux.ld .
```

Let’s edit **main.s** and code it up.

```
/**
 * FILE: main.s
 *
 * DESCRIPTION:
 * This file contains the assembly code for a simple load and store firmware
 * example utilizing the STM32F401CC6 microcontroller.
 *
 * AUTHOR: Kevin Thomas
 * CREATION DATE: July 21, 2023
 * UPDATE Date: July 21, 2023
 *
 * ASSEMBLE AND LINK w/ SYMBOLS:
 * 1. arm-none-eabi-as -g main.s -o main.o
 * 2. arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
 * 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf verify reset exit"
 * ASSEMBLE AND LINK w/o SYMBOLS:
 * 1. arm-none-eabi-as -g main.s -o main.o
 * 2. arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
 * 3. arm-none-eabi-objcopy -O binary --strip-all main.elf main.bin
 * 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.bin 0x08000000 verify reset exit"
 * DEBUG w/ SYMBOLS:
 * 1. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
 * 2. arm-none-eabi-gdb main.elf
 * 3. target remote :3333
 * 4. monitor reset halt
 * 5. l
 * DEBUG w/o SYMBOLS:
 * 1. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
 * 2. arm-none-eabi-gdb main.bin
 * 3. target remote :3333
 * 4. monitor reset halt
 * 5. x/8i $pc
 */

.syntax unified
.cpu cortex-m4
.thumb

/**
 * Provide weak aliases for each Exception handler to the Default_Handler.
 * As they are weak aliases, any function with the same name will override
 * this definition.
 */
.macro weak name
  .global \name
  .weak \name
  .thumb_set \name, Default_Handler
  .word \name
.endm

/**
 * The STM32F401CCUx vector table.  Note that the proper constructs
 * must be placed on this to ensure that it ends up at physical address
 * 0x0000.0000.
 */
.global isr_vector
.section .isr_vector, "a"
.type isr_vector, %object
isr_vector:
  .word _estack
  .word Reset_Handler
   weak NMI_Handler
   weak HardFault_Handler
   weak MemManage_Handler
   weak BusFault_Handler
   weak UsageFault_Handler
```

```
.word 0
.word 0
.word 0
.word 0
 weak SVC_Handler
 weak DebugMon_Handler
.word 0
 weak PendSV_Handler
 weak SysTick_Handler
.word 0
 weak EXTI16_PVD_IRQHandler                             // EXTI Line 16 interrupt /PVD through EXTI line detection interrupt
 weak TAMP_STAMP_IRQHandler                             // Tamper and TimeStamp interrupts through the EXTI line
 weak EXTI22_RTC_WKUP_IRQHandler                        // EXTI Line 22 interrupt /RTC Wakeup interrupt through the EXTI line
 weak FLASH_IRQHandler                                  // FLASH global interrupt
 weak RCC_IRQHandler                                    // RCC global interrupt
 weak EXTI0_IRQHandler                                  // EXTI Line0 interrupt
 weak EXTI1_IRQHandler                                  // EXTI Line1 interrupt
 weak EXTI2_IRQHandler                                  // EXTI Line2 interrupt
 weak EXTI3_IRQHandler                                  // EXTI Line3 interrupt
 weak EXTI4_IRQHandler                                  // EXTI Line4 interrupt
 weak DMA1_Stream0_IRQHandler                           // DMA1 Stream0 global interrupt
 weak DMA1_Stream1_IRQHandler                           // DMA1 Stream1 global interrupt
 weak DMA1_Stream2_IRQHandler                           // DMA1 Stream2 global interrupt
 weak DMA1_Stream3_IRQHandler                           // DMA1 Stream3 global interrupt
 weak DMA1_Stream4_IRQHandler                           // DMA1 Stream4 global interrupt
 weak DMA1_Stream5_IRQHandler                           // DMA1 Stream5 global interrupt
 weak DMA1_Stream6_IRQHandler                           // DMA1 Stream6 global interrupt
 weak ADC_IRQHandler                                    // ADC1 global interrupt
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
 weak EXTI9_5_IRQHandler                                // EXTI Line[9:5] interrupts
 weak TIM1_BRK_TIM9_IRQHandle                           // TIM1 Break interrupt and TIM9 global interrupt
 weak TIM1_UP_TIM10_IRQHandler                          // TIM1 Update interrupt and TIM10 global interrupt
 weak TIM1_TRG_COM_TIM11_IRQHandler                     // TIM1 Trigger and Commutation interrupts and TIM11 global interrupt
 weak TIM1_CC_IRQHandler                                // TIM1 Capture Compare interrupt
 weak TIM2_IRQHandler                                   // TIM2 global interrupt
 weak TIM3_IRQHandler                                   // TIM3 global interrupt
 weak TIM4_IRQHandler                                   // TIM4 global interrupt
 weak I2C1_EV_IRQHandler                                // I2C1 event interrupt
 weak I2C1_ER_IRQHandler                                // I2C1 error interrupt
 weak I2C2_EV_IRQHandler                                // I2C2 event interrupt
 weak I2C2_ER_IRQHandler                                // I2C2 error interrupt
 weak SPI1_IRQHandler                                   // SPI1 global interrupt
 weak SPI2_IRQHandler                                   // SPI2 global interrupt
 weak USART1_IRQHandler                                 // USART1 global interrupt
 weak USART2_IRQHandler                                 // USART2 global interrupt
.word 0                                                 // Reserved
 weak EXTI15_10_IRQHandler                              // EXTI Line[15:10] interrupts
 weak EXTI17_RTC_Alarm_IRQHandler                       // EXTI Line 17 interrupt / RTC Alarms (A and B) through EXTI line interrupt
 weak EXTI18_OTG_FS_WKUP_IRQHandler                     // EXTI Line 18 interrupt / USBUSB OTG FS Wakeup through EXTI line interrupt
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
 weak DMA1_Stream7_IRQHandler                           // DMA1 Stream7 global interrupt
.word 0                                                 // Reserved
 weak SDIO_IRQHandler                                   // SDIO global interrupt
 weak TIM5_IRQHandler                                   // TIM5 global interrupt
 weak SPI3_IRQHandler                                   // SPI3 global interrupt
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
 weak DMA2_Stream0_IRQHandler                           // DMA2 Stream0 global interrupt
 weak DMA2_Stream1_IRQHandler                           // DMA2 Stream1 global interrupt
 weak DMA2_Stream2_IRQHandler                           // DMA2 Stream2 global interrupt
 weak DMA2_Stream3_IRQHandler                           // DMA2 Stream3 global interrupt
 weak DMA2_Stream4_IRQHandler                           // DMA2 Stream4 global interrupt
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
 weak OTG_FS_IRQHandler                                 // USB On The Go FS global interrupt
 weak DMA2_Stream5_IRQHandler                           // DMA2 Stream5 global interrupt
 weak DMA2_Stream6_IRQHandler                           // DMA2 Stream6 global interrupt
 weak DMA2_Stream7_IRQHandler                           // DMA2 Stream7 global interrupt
 weak USART6_IRQHandler                                 // USART6 global interrupt
 weak I2C3_EV_IRQHandler                                // I2C3 event interrupt
 weak I2C3_ER_IRQHandler                                // I2C3 error interrupt
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
.word 0                                                 // Reserved
```

```
  weak SPI4_IRQHandler                                  // SPI4 global interrupt

.section .text

/**
 * @brief  This code is called when processor starts execution.
 *
 *         This is the code that gets called when the processor first
 *         starts execution following a reset event. Only the absolutely
 *         necessary set is performed, after which the application
 *         supplied main() routine is called.
 * @param  None
 * @retval None
 */
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
  LDR   R0, =_estack                                    // load address at end of the stack into R0
  MOV   SP, R0                                          // move address at end of stack into SP
  BL    __start                                         // call function

/**
 * @brief  This code is called when the processor receives and unexpected interrupt.
 *
 *         This is the code that gets called when the processor receives an
 *         unexpected interrupt.  This simply enters an infinite loop, preserving
 *         the system state for examination by a debugger.
 *
 * @param  None
 * @retval None
 */
.type Default_Handler, %function
.global Default_Handler
Default_Handler:
  BKPT                                                  // set processor into debug state
  B.N   Default_Handler                                 // call function, force thumb state

/**
 * @brief  Entry point for initialization and setup of specific functions.
 *
 *         This function is the entry point for initializing and setting up specific functions.
 *         It calls other functions to enable certain features and then enters a loop for further execution.
 *
 * @param  None
 * @retval None
 */
.type __start, %function
__start:
  LDR  R0, =0x86753090                                  // move the literal value of 0x8675309 into R0
  MOVW R0, #0x3090                                      // move the literal value of 0x3090 into the MSBs of R0
  MOVT R0, #0x8675                                      // move the literal value of 0x8675 into the LSBs of R0
  B    .                                                // branch infinite loop
```

We learned about LDR and STR in our last chapter. Today we will cover the MOV instruction.

With our LDR instruction we can do the following.

```
  LDR  R0, =0x86753090                                  // move the literal value of 0x8675309 into R0
```

There is a MOVW instruction which the operand is restricted to 16-bits of immediate data and there is a MOVT instruction which places a 16-bit value in the most significant bits of a register.

If we have a value in hex say, 0x86753090, and we wanted to load this into R0 we would have to do the following.

```
  MOVW R0, #0x3090                                      // move the literal value of 0x3090 into the MSBs of R0
  MOVT R0, #0x8675                                      // move the literal value of 0x8675 into the LSBs of R0
```

Let’s see this in action by first assembling then linking and finally flashing to our MCU.

```
arm-none-eabi-as -g main.s -o main.o
```

```
arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
```

```
openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf verify reset exit"
```

Now let’s fire up our debugger to peek inside!

Terminal 1:

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

Terminal 2:

```
arm-none-eabi-gdb main.elf
```

```
target remote :3333
```

```
monitor reset halt
```

Now that we are inside the firmware, lets set a breakpoint on our __start function which is what is directly executed after the Reset_Handler. Let’s continue and disassemble.

```
(gdb) b __start
Breakpoint 1 at 0x80001a0: file main.s, line 208.
Note: automatically using hardware breakpoints for read-only addresses.
(gdb) c
Continuing.

Breakpoint 1, __start () at main.s:208
208       LDR  R0, =0x86753090                                    // move the literal value of 0x8675309 into R0
```

Let’s disassemble shall we?

```
(gdb) disas
Dump of assembler code for function __start:
=> 0x080001a0 <+0>:     ldr     r0, [pc, #12]   ; (0x80001b0 <__start+16>)
   0x080001a2 <+2>:     movw    r0, #12432      ; 0x3090
   0x080001a6 <+6>:     movt    r0, #34421      ; 0x8675
   0x080001aa <+10>:    b.n     0x80001aa <__start+10>
   0x080001ac <+12>:    lsls    r0, r0, #16
   0x080001ae <+14>:    movs    r0, #0
   0x080001b0 <+16>:    adds    r0, #144        ; 0x90
   0x080001b2 <+18>:    strh    r5, [r6, #50]   ; 0x32
End of assembler dump.
```

Let’s look at what is inside R0, we should know by now ;)

```
(gdb) i r r0
r0             0x20000400          536871936
```

This of course is the end of stack which was completed in the Reset_Handler.

Let’s step again and see what value goes into R0.

```
(gdb) si
209       MOVW R0, #0x3090                                        // move the literal value of 0x3090 into the MSBs of R0
(gdb) i r r0
r0             0x86753090          -2039140208
```

Let’s step again and see what value is in R0.

```
(gdb) si
210       MOVT R0, #0x8675                                          // move the literal value of 0x8675 into the LSBs of R0
(gdb) i r r0
r0             0x3090              12432
```

We can see that 0x3090 was moved into the least significant bits as expected. Keep in mind 0x3090 is the same as 0x00003090.

Step again shall we!

```
(gdb) si
211       B    .                                                    // branch infinite loop
(gdb) i r r0
r0             0x86753090          -2039140208
```

Now we see the full value inside R0.

In our next lesson we will cover conditional execution.

Chapter 10: Conditional Execution

Let’s talk about flags.

Let’s get our project setup below and copy over our template.

```
cd stm32f401ccu6-projects
mkdir 0x0004-conditional_execution
cd 0x0004-conditional_execution
cp ..\0x0001-template\main.s .
cp ..\0x0001-template\stm32f401ccux.ld .
```

Let’s edit **main.s** and code it up.

```
/**
 * FILE: main.s
 *
 * DESCRIPTION:
 * This file contains the assembly code for a simple conditional
 * execution utilizing the STM32F401CC6 microcontroller.
 *
 * AUTHOR: Kevin Thomas
 * CREATION DATE: July 22, 2023
 * UPDATE Date: July 22, 2023
 *
 * ASSEMBLE AND LINK w/ SYMBOLS:
 * 1. arm-none-eabi-as -g main.s -o main.o
 * 2. arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
 * 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf verify reset exit"
 * ASSEMBLE AND LINK w/o SYMBOLS:
 * 1. arm-none-eabi-as -g main.s -o main.o
 * 2. arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
 * 3. arm-none-eabi-objcopy -O binary --strip-all main.elf main.bin
 * 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.bin 0x08000000 verify reset exit"
 * DEBUG w/ SYMBOLS:
 * 1. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
 * 2. arm-none-eabi-gdb main.elf
 * 3. target remote :3333
 * 4. monitor reset halt
 * 5. l
 * DEBUG w/o SYMBOLS:
 * 1. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
 * 2. arm-none-eabi-gdb main.bin
 * 3. target remote :3333
 * 4. monitor reset halt
 * 5. x/8i $pc
 */

.syntax unified
.cpu cortex-m4
.thumb

/**
 * Provide weak aliases for each Exception handler to the Default_Handler.
 * As they are weak aliases, any function with the same name will override
 * this definition.
 */
.macro weak name
  .global \name
  .weak \name
  .thumb_set \name, Default_Handler
  .word \name
.endm

/**
 * The STM32F401CCUx vector table.  Note that the proper constructs
 * must be placed on this to ensure that it ends up at physical address
 * 0x0000.0000.
 */
.global isr_vector
.section .isr_vector, "a"
.type isr_vector, %object
isr_vector:
  .word _estack
  .word Reset_Handler
   weak NMI_Handler
   weak HardFault_Handler
   weak MemManage_Handler
   weak BusFault_Handler
   weak UsageFault_Handler
  .word 0
  .word 0
```

```
.word 0
.word 0
 weak SVC_Handler
 weak DebugMon_Handler
.word 0
 weak PendSV_Handler
 weak SysTick_Handler
.word 0
 weak EXTI16_PVD_IRQHandler                        // EXTI Line 16 interrupt /PVD through EXTI line detection interrupt
 weak TAMP_STAMP_IRQHandler                        // Tamper and TimeStamp interrupts through the EXTI line
 weak EXTI22_RTC_WKUP_IRQHandler                   // EXTI Line 22 interrupt /RTC Wakeup interrupt through the EXTI line
 weak FLASH_IRQHandler                             // FLASH global interrupt
 weak RCC_IRQHandler                               // RCC global interrupt
 weak EXTI0_IRQHandler                             // EXTI Line0 interrupt
 weak EXTI1_IRQHandler                             // EXTI Line1 interrupt
 weak EXTI2_IRQHandler                             // EXTI Line2 interrupt
 weak EXTI3_IRQHandler                             // EXTI Line3 interrupt
 weak EXTI4_IRQHandler                             // EXTI Line4 interrupt
 weak DMA1_Stream0_IRQHandler                      // DMA1 Stream0 global interrupt
 weak DMA1_Stream1_IRQHandler                      // DMA1 Stream1 global interrupt
 weak DMA1_Stream2_IRQHandler                      // DMA1 Stream2 global interrupt
 weak DMA1_Stream3_IRQHandler                      // DMA1 Stream3 global interrupt
 weak DMA1_Stream4_IRQHandler                      // DMA1 Stream4 global interrupt
 weak DMA1_Stream5_IRQHandler                      // DMA1 Stream5 global interrupt
 weak DMA1_Stream6_IRQHandler                      // DMA1 Stream6 global interrupt
 weak ADC_IRQHandler                               // ADC1 global interrupt
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
 weak EXTI9_5_IRQHandler                           // EXTI Line[9:5] interrupts
 weak TIM1_BRK_TIM9_IRQHandle                      // TIM1 Break interrupt and TIM9 global interrupt
 weak TIM1_UP_TIM10_IRQHandler                     // TIM1 Update interrupt and TIM10 global interrupt
 weak TIM1_TRG_COM_TIM11_IRQHandler                // TIM1 Trigger and Commutation interrupts and TIM11 global interrupt
 weak TIM1_CC_IRQHandler                           // TIM1 Capture Compare interrupt
 weak TIM2_IRQHandler                              // TIM2 global interrupt
 weak TIM3_IRQHandler                              // TIM3 global interrupt
 weak TIM4_IRQHandler                              // TIM4 global interrupt
 weak I2C1_EV_IRQHandler                           // I2C1 event interrupt
 weak I2C1_ER_IRQHandler                           // I2C1 error interrupt
 weak I2C2_EV_IRQHandler                           // I2C2 event interrupt
 weak I2C2_ER_IRQHandler                           // I2C2 error interrupt
 weak SPI1_IRQHandler                              // SPI1 global interrupt
 weak SPI2_IRQHandler                              // SPI2 global interrupt
 weak USART1_IRQHandler                            // USART1 global interrupt
 weak USART2_IRQHandler                            // USART2 global interrupt
.word 0                                            // Reserved
 weak EXTI15_10_IRQHandler                         // EXTI Line[15:10] interrupts
 weak EXTI17_RTC_Alarm_IRQHandler                  // EXTI Line 17 interrupt / RTC Alarms (A and B) through EXTI line interrupt
 weak EXTI18_OTG_FS_WKUP_IRQHandler                // EXTI Line 18 interrupt / USBUSB OTG FS Wakeup through EXTI line interrupt
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
 weak DMA1_Stream7_IRQHandler                      // DMA1 Stream7 global interrupt
.word 0                                            // Reserved
 weak SDIO_IRQHandler                              // SDIO global interrupt
 weak TIM5_IRQHandler                              // TIM5 global interrupt
 weak SPI3_IRQHandler                              // SPI3 global interrupt
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
 weak DMA2_Stream0_IRQHandler                      // DMA2 Stream0 global interrupt
 weak DMA2_Stream1_IRQHandler                      // DMA2 Stream1 global interrupt
 weak DMA2_Stream2_IRQHandler                      // DMA2 Stream2 global interrupt
 weak DMA2_Stream3_IRQHandler                      // DMA2 Stream3 global interrupt
 weak DMA2_Stream4_IRQHandler                      // DMA2 Stream4 global interrupt
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
 weak OTG_FS_IRQHandler                            // USB On The Go FS global interrupt
 weak DMA2_Stream5_IRQHandler                      // DMA2 Stream5 global interrupt
 weak DMA2_Stream6_IRQHandler                      // DMA2 Stream6 global interrupt
 weak DMA2_Stream7_IRQHandler                      // DMA2 Stream7 global interrupt
 weak USART6_IRQHandler                            // USART6 global interrupt
 weak I2C3_EV_IRQHandler                           // I2C3 event interrupt
 weak I2C3_ER_IRQHandler                           // I2C3 error interrupt
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
.word 0                                            // Reserved
 weak SPI4_IRQHandler                              // SPI4 global interrupt
```

```
.section .text

/**
 * @brief  This code is called when processor starts execution.
 *
 *         This is the code that gets called when the processor first
 *         starts execution following a reset event. Only the absolutely
 *         necessary set is performed, after which the application
 *         supplied main() routine is called.
 * @param  None
 * @retval None
 */
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
  LDR   R0, =_estack                             // load address at end of the stack into R0
  MOV   SP, R0                                   // move address at end of stack into SP
  BL    __start                                  // call function

/**
 * @brief  This code is called when the processor receives and unexpected interrupt.
 *
 *         This is the code that gets called when the processor receives an
 *         unexpected interrupt.  This simply enters an infinite loop, preserving
 *         the system state for examination by a debugger.
 *
 * @param  None
 * @retval None
 */
.type Default_Handler, %function
.global Default_Handler
Default_Handler:
  BKPT                                           // set processor into debug state
  B.N   Default_Handler                          // call function, force thumb state

/**
 * @brief  Entry point for initialization and setup of specific functions.
 *
 *         This function is the entry point for initializing and setting up specific functions.
 *         It calls other functions to enable certain features and then enters a loop for further execution.
 *
 * @param  None
 * @retval None
 */
.type __start, %function
__start:
  MOV  R0, #0x42                                 // move 0x42 into R0
  MOV  R1, #0x42                                 // move 0x42 into R1
  CMP  R0, R1                                    // compare R0 - R1
  BEQ  Equal                                     // branch if equal

Equal:
  NOP                                            // no operation instruction

  MOV  R0, #0x43                                 // move 0x43 into R0
  MOV  R1, #0x42                                 // move 0x42 into R1
  CMP  R0, R1                                    // compare R0 - R1
  BGT  Greater                                   // branch if greater than

Greater:
  NOP                                            // no operation instruction

  MOV  R0, #0x42                                 // move 0x42 into R0
  MOV  R1, #0x43                                 // move 0x43 into R1
  CMP  R0, R1                                    // compare R0 - R1
  BLT  Less                                      // branch if less than

Less:
  NOP                                            // no operation instruction

  LDR  R0, =0x40023830                           // load address of RCC_AHB1ENR register
  LDR  R1, [R0]                                  // load value inside RCC_AHB1ENR register
  ORR  R1, #(1<<0)                               // set the GPIOAEN bit
  TST  R1, #(1<<0)                               // test if bit 0 is set
  BNE  Bit_Set                                   // branch if not equal

Bit_Set:
  NOP                                            // no operation instruction

  LDR  R0, =0x40023830                           // load address of RCC_AHB1ENR register
  LDR  R1, [R0]                                  // load value inside RCC_AHB1ENR register
  BIC  R1, #(1<<0)                               // clear the GPIOAEN bit
  TST  R1, #(1<<0)                               // test if bit 0 is set
  BEQ  Bit_Not_Set                               // branch if equal

Bit_Not_Set:
  NOP                                            // no operation instruction

Loop:
  B    .                                         // branch infinite loop
```

Let’s break this down one example at a time.

```
MOV  R0, #0x42                                    // move 0x42 into R0
MOV  R1, #0x42                                    // move 0x42 into R1
CMP  R0, R1                                       // compare R0 - R1
BEQ  Equal                                        // branch if equal
```

In any language we need the ability to make conditional execution so that we can control program flow. Here we will break down to check for an equal condition.

We first move 0x42 into R0 and then we move 0x42 into R1 and then we use the CMP, compare instruction, to do a subtraction without actually changing any values and if the result is zero we will branch to the Equal label.

There are flags that will be set as well and when we do our debugging we will see how the status register will be effected as the zero flag will be set and branch appropriately.

```
MOV  R0, #0x43                                    // move 0x43 into R0
MOV  R1, #0x42                                    // move 0x42 into R1
CMP  R0, R1                                       // compare R0 - R1
BGT  Greater                                      // branch if greater than
```

We then move 0x43 into R0 and 0x42 into R1 and do a compare and in this situation, we get a greater than condition so the zero flag will not be set and branch appropriately.

```
MOV  R0, #0x42                                    // move 0x42 into R0
MOV  R1, #0x43                                    // move 0x43 into R1
CMP  R0, R1                                       // compare R0 - R1
BLT  Less                                         // branch if less than
```

We then move 0x42 into R0 and 0x43 into R1 and do a compare and in this situation we get a less than condition so the negative flag will be set.

It is also important that we be able to test individual bits within a register to base conditional execution on. In this below case we will test if the bit is set and branch appropriately.

```
LDR  R0, =0x40023830                              // load address of RCC_AHB1ENR register
LDR  R1, [R0]                                     // load value inside RCC_AHB1ENR register
ORR  R1, #(1<<0)                                  // set the GPIOAEN bit
TST  R1, #(1<<0)                                  // test if bit 0 is set
BNE  Bit_Set                                      // branch if not equal
```

Here we see that we set the bit with the ORR instruction that bit 0 is set to 1 without disturbing any other bits. We then test to see if that bit is set and if it is it will branch not equal.

Finally we test a situation where a bit is clear or not set.

```
LDR  R0, =0x40023830                              // load address of RCC_AHB1ENR register
LDR  R1, [R0]                                     // load value inside RCC_AHB1ENR register
BIC  R1, #(1<<0)                                  // clear the GPIOAEN bit
TST  R1, #(1<<0)                                  // test if bit 0 is set
BEQ  Bit_Not_Set                                  // branch if equal
```

In this situation we first bit clear, BIC, bit 0 to make sure it is 0 without disturbing any other bits. When we test we will see that it is in fact equal and the zero flag will be set and branch appropriately.

Let’s see this in action by first assembling then linking and finally flashing to our MCU.

```
arm-none-eabi-as -g main.s -o main.o

arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld

openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf verify reset exit"
```

Let’s debug!

Terminal 1:

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

Terminal 2:

```
arm-none-eabi-gdb main.elf

target remote :3333

monitor reset halt
```

Now that we are inside the firmware, lets set a breakpoint on our __start function which is what is directly executed after the Reset_Handler. Let’s continue and disassemble.

```
(gdb) b __start
Breakpoint 1 at 0x80001a0: file main.s, line 208.
Note: automatically using hardware breakpoints for read-only addresses.
(gdb) c
Continuing.

Breakpoint 1, __start () at main.s:208
208       MOV  R0, #0x42                                       // move 0x42 into R0
(gdb) disas
Dump of assembler code for function __start:
=> 0x080001a0 <+0>:     mov.w   r0, #66 ; 0x42
   0x080001a4 <+4>:     mov.w   r1, #66 ; 0x42
   0x080001a8 <+8>:     cmp     r0, r1
   0x080001aa <+10>:    beq.n   0x80001ac <Equal>
End of assembler dump.
```

We know what R0 currently has after coming out of the Reset_Handler so lets step into three times as we also know what R0 and R1 will have.

```
(gdb) si
209       MOV  R1, #0x42                                       // move 0x42 into R1
(gdb) si
210       CMP  R0, R1                                          // compare R0 - R1
(gdb) si
211       BEQ  Equal                                           // branch if equal
```

Let’s check the status of the xPSR.

```
(gdb) p/x $xPSR
$5 = 0x61000000
```

Let’s use a tool like binaryhexconverter.com at https://www.binaryhexconverter.com/hex-to-binary-converter to see what the values are.

```
01100001000000000000000000000000
```

In the ARMv7-M Architecture Reference Manual on page 31 we see the values inside the status register.

bit 31 N flag (negative condition code flag)
bit 30 Z flag (zero condition code flag)
bit 29 C flag (carry condition code flag)
bit 28 V flag (overflow condition code flag)

The Z zero flag is set because the results of the operation are zero.

The C carry flag is set indicating that there is no borrow or underflow from the most significant bit.

Bit 24 in the xPSR is set. This bit is called the T bit. It is set if the processor is currently running in Thumb state. Thumb state is a special mode of the Cortex-M4 processor that allows it to execute Thumb instructions more efficiently.

The T bit is set by the processor automatically when it enters Thumb state. It is cleared by the processor automatically when it exits Thumb state.

The T bit is used by the processor to determine how to decode instructions. If the T bit is set, the processor will decode instructions as Thumb instructions. If the T bit is clear, the processor will decode instructions as ARM instructions.

In this case, the T bit is set because the code that you are running is in Thumb state. You can tell that the code is in Thumb state because the mov.w instruction is a Thumb instruction.

Let’s disas!

```
(gdb) disas
Dump of assembler code for function __start:
   0x080001a0 <+0>:     mov.w   r0, #66 ; 0x42
   0x080001a4 <+4>:     mov.w   r1, #66 ; 0x42
   0x080001a8 <+8>:     cmp     r0, r1
=> 0x080001aa <+10>:    beq.n   0x80001ac <Equal>
End of assembler dump.
```

We should expect that we should branch to the Equal label so lets step into twice.

```
(gdb) si
Equal () at main.s:214
```

```
214       NOP                                                     // no operation instruction
(gdb) si
216       MOV  R0, #0x43                                          // move 0x43 into R0
```

We can continue now to look at the next block of code.

```
(gdb) disas
Dump of assembler code for function Equal:
   0x080001ac <+0>:     nop
=> 0x080001ae <+2>:     mov.w   r0, #67 ; 0x43
   0x080001b2 <+6>:     mov.w   r1, #66 ; 0x42
   0x080001b6 <+10>:    cmp     r0, r1
   0x080001b8 <+12>:    bgt.n   0x80001ba <Greater>
```

Let’s step until we step over the compare and look at the value of the xPSR.

```
(gdb) si
217       MOV  R1, #0x42                                          // move 0x42 into R1
(gdb) si
218       CMP  R0, R1                                             // compare R0 - R1
(gdb) si
219       BGT  Greater                                            // branch if greater than
(gdb) p/x $xPSR
$6 = 0x21000000
```

Let’s examine what our xPSR is using our conversion tool.

```
00100001000000000000000000000000
```

The C carry flag is set indicating that there is no borrow or underflow from the most significant bit.

We also see the T bit or Thumb bit set as well.

When we step we enter into the Greater label as expected.

```
(gdb) si
Greater () at main.s:222
222       NOP                                                     // no operation instruction
```

Let’s examine our next condition.

```
(gdb) si
224       MOV  R0, #0x42                                          // move 0x42 into R0
(gdb) disas
Dump of assembler code for function Greater:
   0x080001ba <+0>:     nop
=> 0x080001bc <+2>:     mov.w   r0, #66 ; 0x42
   0x080001c0 <+6>:     mov.w   r1, #67 ; 0x43
   0x080001c4 <+10>:    cmp     r0, r1
   0x080001c6 <+12>:    blt.n   0x80001c8 <Less>
End of assembler dump.
```

Let’s step a few times and look at the xPSR.

```
(gdb) si
225       MOV  R1, #0x43                                          // move 0x43 into R1
(gdb) si
226       CMP  R0, R1                                             // compare R0 - R1
(gdb) si
227       BLT  Less                                               // branch if less than
(gdb) p/x $xPSR
$7 = 0x81000000
```

Let’s convert the value.

```
10000001000000000000000000000000
```

Here we see the negative N flag set as the result of the compare is a negative and we also see the T bit set as expected.

Let’s continue and prove we will go into the Less label.

```
(gdb) si
Less () at main.s:230
230       NOP                                          // no operation instruction
```

Let’s examine our next example with bit set.

```
(gdb) si
232       LDR  R0, =0x40023830                          // load address of RCC_AHB1ENR register
(gdb) disas
Dump of assembler code for function Less:
   0x080001c8 <+0>:     nop
=> 0x080001ca <+2>:     ldr     r0, [pc, #36]   ; (0x80001f0 <Loop+6>)
   0x080001cc <+4>:     ldr     r1, [r0, #0]
   0x080001ce <+6>:     orr.w   r1, r1, #1
   0x080001d2 <+10>:    tst.w   r1, #1
   0x080001d6 <+14>:    bne.n   0x80001d8 <Bit_Set>
```

Let’s step a few times and once again examine the xPSR.

```
(gdb) si
233       LDR  R1, [R0]                                 // load value inside RCC_AHB1ENR register
(gdb) si
234       ORR  R1, #(1<<0)                              // set the GPIOAEN bit
(gdb) si
235       TST  R1, #(1<<0)                              // test if bit 0 is set
(gdb) si
236       BNE  Bit_Set                                  // branch if not equal
(gdb) p/x $xPSR
$8 = 0x1000000
```

Let’s look at the xPSR conversion.

```
0001000000000000000000000000
```

The TST instruction is used to test bits in register R1 using a bitmask (1<<0). The TST instruction performs a bitwise AND operation between the value in R1 and the bitmask, updating the condition flags based on the result.

Now, let's analyze the value in register R1 after the ORR instruction:

Before the ORR instruction, the value in R1 was loaded from the memory address pointed to by R0. The specific value depends on the content of the RCC_AHB1ENR register, but it's assumed that bit 0 (GPIOAEN) is initially cleared (set to 0).

After the ORR instruction, bit 0 of R1 is set to 1 using the ORR operation with the bitmask (1<<0). This sets the GPIOAEN bit to enable GPIO port A.

Now, let's look at the TST instruction:

The TST instruction performs a bitwise AND between the value in R1 (after the ORR operation) and the bitmask (1<<0). This results in R1 & (1<<0), which is 0x1 & 0x1, equal to 0x1.

Since the result of the AND operation is not zero (non-zero), the Zero (Z) flag in the xPSR register will be cleared, and the V (Overflow) flag will be set to 1. The V flag indicates that the result of the AND operation does not fit in a single unsigned bit, and there is an overflow in this case.

So, the V flag is set because the TST instruction detects that bit 0 of R1 is set after the ORR operation. The value 0x1000000 in xPSR indicates that the V flag is set.

We notice the T bit was not set as ORR is not a thumb instruction however it is an ARM instruction.

BNE (Branch if Not Equal):

BNE performs the branch if the Zero (Z) flag is clear (0).
The Z flag is set (1) when the result of a previous instruction was zero.

So, BNE branches when the result of the previous instruction is non-zero, indicating inequality.

<u>In other words, BNE checks if the tested bit is set to 1.</u>

BEQ (Branch if Equal):

BEQ performs the branch if the Zero (Z) flag is set (1).
The Z flag is set (1) when the result of a previous instruction was zero.

So, BEQ branches when the result of the previous instruction is zero, indicating equality.

<u>In other words, BEQ checks if the tested bit is set to 0.</u>

Lets continue and prove we go into the Bit_Set label.

```
(gdb) si
Bit_Set () at main.s:239
239       NOP                                                   // no operation instruction
```

Let’s examine our final block.

```
(gdb) disas
Dump of assembler code for function Bit_Set:
   0x080001d8 <+0>:     nop
=> 0x080001da <+2>:     ldr     r0, [pc, #20]   ; (0x80001f0 <Loop+6>)
   0x080001dc <+4>:     ldr     r1, [r0, #0]
   0x080001de <+6>:     bic.w   r1, r1, #1
   0x080001e2 <+10>:    tst.w   r1, #1
   0x080001e6 <+14>:    beq.n   0x80001e8 <Bit_Not_Set>
End of assembler dump.
```

Let’s step and see the xPSR.

```
(gdb) si
242       LDR  R1, [R0]                                    // load value inside RCC_AHB1ENR register
(gdb) si
243       BIC  R1, #(1<<0)                                 // clear the GPIOAEN bit
(gdb) si
244       TST  R1, #(1<<0)                                 // test if bit 0 is set
(gdb) si
245       BEQ  Bit_Not_Set                                 // branch if equal
(gdb) p/x $xPSR
$9 = 0x41000000
```

Let’s look at the xPSR conversion.

```
01000001000000000000000000000000
```

Here we see the Z and T flags set which should be obvious as we cleared the bit. The BIC is a Thumb instruction as well.

Let’s show that we continue to the Bit_Not_Set.

```
(gdb) si
Bit_Not_Set () at main.s:248
248       NOP
```

I realize this is a lot of work but this is how one masters the MCU!

In our final chapter we will cover finish up our learning with a cool demo!

We will cover functions, UART and interrupts as well.

Chapter 11: Functions, Interrupts, UART & STUXNET Simulation!

Let’s talk about functions and interrupts and for fun, a STUXNET industrial control system mock hack with both UART simulating direct into a larger system and UART with bluetooth to demonstrate IoT!

Let’s get our project setup below and copy over our template. We will be using the STM32 NUCLEO-F401RE dev board in this chapter as it has a 5-volt rail line.

```
cd stm32f401ccu6-projects
mkdir 0x0006-int-stuxnet
cd 0x0006-int-stuxnet
cp ..\0x0001-template\main.s .
cp ..\0x0001-template\stm32f401ccux.ld .
```

Let’s edit **main.s** and code it up.

```
/**
 * FILE: main.s
 *
 * DESCRIPTION:
 * This file contains the assembly code for interrupts and a STUXNET simulation
 * utilizing the STM32F401 Nucleo-64 microcontroller.
 *
 * AUTHOR: Kevin Thomas
 * CREATION DATE: September 1, 2023
 * UPDATE Date: September 4, 2023
 *
 * ASSEMBLE AND LINK w/ SYMBOLS:
 * 1. arm-none-eabi-as -g main.s -o main.o
 * 2. arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
 * 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf verify reset exit"
 * ASSEMBLE AND LINK w/o SYMBOLS:
 * 1. arm-none-eabi-as -g main.s -o main.o
 * 2. arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
 * 3. arm-none-eabi-objcopy -O binary --strip-all main.elf main.bin
 * 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.bin 0x08000000 verify reset exit"
 * DEBUG w/ SYMBOLS:
 * 1. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
 * 2. arm-none-eabi-gdb main.elf
 * 3. target remote :3333
 * 4. monitor reset halt
 * 5. l
 * DEBUG w/o SYMBOLS:
 * 1. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg
 * 2. arm-none-eabi-gdb main.bin
 * 3. target remote :3333
 * 4. monitor reset halt
 * 5. x/8i $pc
 */

.syntax unified
.cpu cortex-m4
.thumb

/**
 * Provide weak aliases for each Exception handler to the Default_Handler.
 * As they are weak aliases, any function with the same name will override
 * this definition.
 */
.macro weak name
  .global \name
  .weak \name
  .thumb_set \name, Default_Handler
  .word \name
.endm

/**
 * The STM32F401CCUx vector table.  Note that the proper constructs
 * must be placed on this to ensure that it ends up at physical address
 * 0x00000000.
 */
.global isr_vector
.section .isr_vector, "a"
```

```
.type isr_vector, %object
isr_vector:
  .word _estack
  .word Reset_Handler
   weak NMI_Handler
   weak HardFault_Handler
   weak MemManage_Handler
   weak BusFault_Handler
   weak UsageFault_Handler
  .word 0
  .word 0
  .word 0
  .word 0
   weak SVC_Handler
   weak DebugMon_Handler
  .word 0
   weak PendSV_Handler
   weak SysTick_Handler
  .word 0
   weak EXTI16_PVD_IRQHandler                        // EXTI Line 16 interrupt /PVD through EXTI line detection interrupt
   weak TAMP_STAMP_IRQHandler                        // Tamper and TimeStamp interrupts through the EXTI line
   weak EXTI22_RTC_WKUP_IRQHandler                   // EXTI Line 22 interrupt /RTC Wakeup interrupt through the EXTI line
   weak FLASH_IRQHandler                             // FLASH global interrupt
   weak RCC_IRQHandler                               // RCC global interrupt
   weak EXTI0_IRQHandler                             // EXTI Line0 interrupt
   weak EXTI1_IRQHandler                             // EXTI Line1 interrupt
   weak EXTI2_IRQHandler                             // EXTI Line2 interrupt
   weak EXTI3_IRQHandler                             // EXTI Line3 interrupt
   weak EXTI4_IRQHandler                             // EXTI Line4 interrupt
   weak DMA1_Stream0_IRQHandler                      // DMA1 Stream0 global interrupt
   weak DMA1_Stream1_IRQHandler                      // DMA1 Stream1 global interrupt
   weak DMA1_Stream2_IRQHandler                      // DMA1 Stream2 global interrupt
   weak DMA1_Stream3_IRQHandler                      // DMA1 Stream3 global interrupt
   weak DMA1_Stream4_IRQHandler                      // DMA1 Stream4 global interrupt
   weak DMA1_Stream5_IRQHandler                      // DMA1 Stream5 global interrupt
   weak DMA1_Stream6_IRQHandler                      // DMA1 Stream6 global interrupt
   weak ADC_IRQHandler                               // ADC1 global interrupt
  .word 0                                            // Reserved
  .word 0                                            // Reserved
  .word 0                                            // Reserved
  .word 0                                            // Reserved
   weak EXTI9_5_IRQHandler                           // EXTI Line[9:5] interrupts
   weak TIM1_BRK_TIM9_IRQHandle                      // TIM1 Break interrupt and TIM9 global interrupt
   weak TIM1_UP_TIM10_IRQHandler                     // TIM1 Update interrupt and TIM10 global interrupt
   weak TIM1_TRG_COM_TIM11_IRQHandler                // TIM1 Trigger and Commutation interrupts and TIM11 global interrupt
   weak TIM1_CC_IRQHandler                           // TIM1 Capture Compare interrupt
   weak TIM2_IRQHandler                              // TIM2 global interrupt
   weak TIM3_IRQHandler                              // TIM3 global interrupt
   weak TIM4_IRQHandler                              // TIM4 global interrupt
   weak I2C1_EV_IRQHandler                           // I2C1 event interrupt
   weak I2C1_ER_IRQHandler                           // I2C1 error interrupt
   weak I2C2_EV_IRQHandler                           // I2C2 event interrupt
   weak I2C2_ER_IRQHandler                           // I2C2 error interrupt
   weak SPI1_IRQHandler                              // SPI1 global interrupt
   weak SPI2_IRQHandler                              // SPI2 global interrupt
   weak USART1_IRQHandler                            // USART1 global interrupt
   weak USART2_IRQHandler                            // USART2 global interrupt
  .word 0                                            // Reserved
  .word EXTI15_10_IRQHandler                         // EXTI Line[15:10] interrupts
   weak EXTI17_RTC_Alarm_IRQHandler                  // EXTI Line 17 interrupt / RTC Alarms (A and B) through EXTI line interrupt
   weak EXTI18_OTG_FS_WKUP_IRQHandler                // EXTI Line 18 interrupt / USBUSB OTG FS Wakeup through EXTI line interrupt
  .word 0                                            // Reserved
  .word 0                                            // Reserved
  .word 0                                            // Reserved
  .word 0                                            // Reserved
   weak DMA1_Stream7_IRQHandler                      // DMA1 Stream7 global interrupt
  .word 0                                            // Reserved
   weak SDIO_IRQHandler                              // SDIO global interrupt
   weak TIM5_IRQHandler                              // TIM5 global interrupt
   weak SPI3_IRQHandler                              // SPI3 global interrupt
  .word 0                                            // Reserved
  .word 0                                            // Reserved
  .word 0                                            // Reserved
  .word 0                                            // Reserved
   weak DMA2_Stream0_IRQHandler                      // DMA2 Stream0 global interrupt
   weak DMA2_Stream1_IRQHandler                      // DMA2 Stream1 global interrupt
   weak DMA2_Stream2_IRQHandler                      // DMA2 Stream2 global interrupt
   weak DMA2_Stream3_IRQHandler                      // DMA2 Stream3 global interrupt
   weak DMA2_Stream4_IRQHandler                      // DMA2 Stream4 global interrupt
  .word 0                                            // Reserved
  .word 0                                            // Reserved
  .word 0                                            // Reserved
  .word 0                                            // Reserved
  .word 0                                            // Reserved
  .word 0                                            // Reserved
   weak OTG_FS_IRQHandler                            // USB On The Go FS global interrupt
   weak DMA2_Stream5_IRQHandler                      // DMA2 Stream5 global interrupt
   weak DMA2_Stream6_IRQHandler                      // DMA2 Stream6 global interrupt
   weak DMA2_Stream7_IRQHandler                      // DMA2 Stream7 global interrupt
   weak USART6_IRQHandler                            // USART6 global interrupt
   weak I2C3_EV_IRQHandler                           // I2C3 event interrupt
   weak I2C3_ER_IRQHandler                           // I2C3 error interrupt
  .word 0                                            // Reserved
```

```
  .word 0                                                   // Reserved
  .word 0                                                   // Reserved
  .word 0                                                   // Reserved
  .word 0                                                   // Reserved
  .word 0                                                   // Reserved
  .word 0                                                   // Reserved
  .word 0                                                   // Reserved
  .word 0                                                   // Reserved
  .word 0                                                   // Reserved
   weak SPI4_IRQHandler                                     // SPI4 global interrupt

.section .text

/**
 * @brief   This code is called when processor starts execution.
 *
 * @details This is the code that gets called when the processor first
 *          starts execution following a reset event. Only the absolutely
 *          necessary set is performed, after which the application
 *          supplied __start routine is called.
 *
 * @param   None
 * @retval  None
 */
.type Reset_Handler, %function
.global Reset_Handler
Reset_Handler:
  LDR   R0, =_estack                                        // load address at end of the stack into R0
  MOV   SP, R0                                              // move address at end of stack into SP
  BL    __start                                             // call function

/**
 * @brief   This code is called when the processor receives and unexpected interrupt.
 *
 * @details This is the code that gets called when the processor receives an
 *          unexpected interrupt.  This simply enters an infinite loop, preserving
 *          the system state for examination by a debugger.
 *
 * @param   None
 * @retval  None
 */
.type Default_Handler, %function
.global Default_Handler
Default_Handler:
  BKPT                                                      // set processor into debug state
  B.N   Default_Handler                                     // call function, force thumb state

/**
 * @brief   This code is called when the interrupt handler for the EXTI lines 15 to 10
 *          is triggered.
 *
 * @details This is the interrupt handler function for EXTI lines 15 to 10. It is
 *          triggered when an interrupt request is received on any of these lines.
 *          The function checks if the interrupt was caused by line 13 (PR13 bit),
 *          and if so, it sets the corresponding bit in the EXTI_PR register to
 *          acknowledge the interrupt. After that, it calls the EXTI_Callback function.
 *
 * @param   None
 * @retval  None
 */
.section .text.EXTI15_10_IRQHandler
.weak EXTI15_10_IRQHandler
.type EXTI15_10_IRQHandler, %function
EXTI15_10_IRQHandler:
  PUSH  {LR}                                                // push return address onto stack
  LDR   R0, =0x40013C14                                     // load the address of EXTI_PR register
  LDR   R1, [R0]                                            // load value inside EXTI_PR register
  TST   R1, #(1<<13)                                        // read the PR13 bit, if 0, then BEQ
  BEQ   .PR13_0                                             // branch if equal
  ORR   R1, #(1<<13)                                        // set the PR13 bit
  STR   R1, [R0]                                            // store value inside R1 into R0
  BL    EXTI_Callback                                       // call function
.PR13_0:
  POP   {LR}                                                // pop return address from stack
  BX    LR                                                  // return to caller

/**
 * @brief   Entry point for initialization and setup of specific functions.
 *
 * @details This function is the entry point for initializing and setting up specific functions.
 *          It calls other functions to enable certain features and then enters a loop for further execution.
 *
 * @param   None
 * @retval  None
 */
__start:
  BL    GPIOA_Enable                                        // call function
  BL    GPIOC_Enable                                        // call function
  BL    GPIOA_PA9_Alt_Function_Mode_Enable                  // call function
  BL    GPIOA_PA10_Alt_Function_Mode_Enable                 // call function
  BL    GPIOC_PC0_General_Purpose_Output_Mode_Enable        // call function
  BL    GPIOC_PC1_General_Purpose_Output_Mode_Enable        // call function
  BL    GPIOC_PC2_General_Purpose_Output_Mode_Enable        // call function
```

```
  BL    GPIOC_PC3_General_Purpose_Output_Mode_Enable       // call function
  BL    USART1_Enable                                      // call function
  BL    GPIOC_PC13_EXTI_Init                               // call function
  B     .                                                  // branch infinite loop

/**
 * @brief   Enables the GPIOA peripheral by setting the corresponding RCC_AHB1ENR bit.
 *
 * @details This function enables the GPIOA peripheral by setting the corresponding
 *          RCC_AHB1ENR bit.  It loads the address of the RCC_AHB1ENR register, retrieves
 *          the current value of the register, sets the GPIOAEN bit, and stores the
 *          updated value back into the register.
 *
 * @param   None
 * @retval  None
 */
GPIOA_Enable:
  LDR   R0, =0x40023830                                    // load address of RCC_AHB1ENR register
  LDR   R1, [R0]                                           // load value inside RCC_AHB1ENR register
  ORR   R1, #(1<<0)                                        // set the GPIOAEN bit
  STR   R1, [R0]                                           // store value into RCC_AHB1ENR register
  BX    LR                                                 // return to caller

/**
 * @brief   Enables the GPIOC peripheral by setting the corresponding RCC_AHB1ENR bit.
 *
 * @details This function enables the GPIOC peripheral by setting the corresponding
 *          RCC_AHB1ENR bit.  It loads the address of the RCC_AHB1ENR register, retrieves
 *          the current value of the register, sets the GPIOCEN bit, and stores the
 *          updated value back into the register.
 *
 * @param   None
 * @retval  None
 */
GPIOC_Enable:
  LDR   R0, =0x40023830                                    // load address of RCC_AHB1ENR register
  LDR   R1, [R0]                                           // load value inside RCC_AHB1ENR register
  ORR   R1, #(1<<2)                                        // set the GPIOCEN bit
  STR   R1, [R0]                                           // store value into RCC_AHB1ENR register
  BX    LR                                                 // return to caller

/**
 * @brief   Configures GPIOA pin 9 to operate in alternate function mode.
 *
 * @details This function configures GPIOA pin 9 to operate in alternate function mode.
 *          It modifies the GPIOA_MODER and GPIOA_AFRH registers to set the necessary bits
 *          for alternate function mode on pin 9. The MODER9 bit is set to select alternate
 *          function mode, and the AFRH9 bits are set to specify the desired alternate function.
 *
 * @param   None
 * @retval  None
 */
GPIOA_PA9_Alt_Function_Mode_Enable:
  LDR   R0, =0x40020000                                    // load address of GPIOA_MODER register
  LDR   R1, [R0]                                           // load value inside GPIOA_MODER register
  ORR   R1, #(1<<19)                                       // set the MODER9 bit
  AND   R1, #~(1<<18)                                      // clear the MODER9 bit
  STR   R1, [R0]                                           // store value into GPIOA_MODER register
  LDR   R0, =0x40020024                                    // load address of GPIOA_AFRH register
  LDR   R1, [R0]                                           // load value inside GPIOA_AFRH register
  AND   R1, #~(1<<7)                                       // clear the AFRH9 bit
  ORR   R1, #(1<<6)                                        // set the AFRH9 bit
  ORR   R1, #(1<<5)                                        // set the AFRH9 bit
  ORR   R1, #(1<<4)                                        // set the AFRH9 bit
  STR   R1, [R0]                                           // store value into GPIOA_AFRH register
  BX    LR                                                 // return to caller

/**
 * @brief   Configures GPIOA pin 10 to operate in alternate function mode.
 *
 * @details This function configures GPIOA pin 10 to operate in alternate function mode.
 *          It modifies the GPIOA_MODER and GPIOA_AFRH registers to set the necessary bits
 *          for alternate function mode on pin 10. The MODER10 bit is set to select alternate
 *          function mode, and the AFRH9 bits are set to specify the desired alternate function.
 *
 * @param   None
 * @retval  None
 */
GPIOA_PA10_Alt_Function_Mode_Enable:
  LDR   R0, =0x40020000                                    // load address of GPIOA_MODER register
  LDR   R1, [R0]                                           // load value inside GPIOA_MODER register
  ORR   R1, #(1<<21)                                       // set the MODER10 bit
  AND   R1, #~(1<<20)                                      // clear the MODER10 bit
  STR   R1, [R0]                                           // store value into GPIOA_MODER register
  LDR   R0, =0x40020024                                    // load address of GPIOA_AFRH register
  LDR   R1, [R0]                                           // load value inside GPIOA_AFRH register
  AND   R1, #~(1<<11)                                      // clear the AFRH10 bit
  ORR   R1, #(1<<10)                                       // set the AFRH10 bit
  ORR   R1, #(1<<9)                                        // set the AFRH10 bit
  ORR   R1, #(1<<8)                                        // set the AFRH10 bit
  STR   R1, [R0]                                           // store value into GPIOA_AFRH register
  BX    LR                                                 // return to caller
```

```
/**
 * @brief   Configures GPIOC pin 0 to operate in general purpose output mode.
 *
 * @details This function configures GPIOC pin 0 to operate in general purpose output mode.
 *          It modifies the GPIOC_MODER to set the necessary bits for general purpose output mode
 *          on pin 0. The MODER0 bit is set to select general purpose output mode.
 *
 * @param   None
 * @retval  None
 */
GPIOC_PC0_General_Purpose_Output_Mode_Enable:
  LDR   R0, =0x40020800                                    // load address of GPIOC_MODER register
  LDR   R1, [R0]                                           // load value inside GPIOC_MODER register
  AND   R1, #~(1<<1)                                       // clear the MODER0 bit
  ORR   R1, #(1<<0)                                        // set the MODER0 bit
  STR   R1, [R0]                                           // store value into GPIOC_MODER register
  BX    LR                                                 // return to caller

/**
 * @brief   Configures GPIOC pin 1 to operate in general purpose output mode.
 *
 * @details This function configures GPIOC pin 1 to operate in general purpose output mode.
 *          It modifies the GPIOC_MODER to set the necessary bits for general purpose output mode
 *          on pin 1. The MODER1 bit is set to select general purpose output mode.
 *
 * @param   None
 * @retval  None
 */
GPIOC_PC1_General_Purpose_Output_Mode_Enable:
  LDR   R0, =0x40020800                                    // load address of GPIOC_MODER register
  LDR   R1, [R0]                                           // load value inside GPIOC_MODER register
  AND   R1, #~(1<<3)                                       // clear the MODER1 bit
  ORR   R1, #(1<<2)                                        // set the MODER1 bit
  STR   R1, [R0]                                           // store value into GPIOC_MODER register
  BX    LR                                                 // return to caller

/**
 * @brief   Configures GPIOC pin 2 to operate in general purpose output mode.
 *
 * @details This function configures GPIOC pin 2 to operate in general purpose output mode.
 *          It modifies the GPIAC_MODER to set the necessary bits for general purpose output mode
 *          on pin 2. The MODER2 bit is set to select general purpose output mode.
 *
 * @param   None
 * @retval  None
 */
GPIOC_PC2_General_Purpose_Output_Mode_Enable:
  LDR   R0, =0x40020800                                    // load address of GPIOC_MODER register
  LDR   R1, [R0]                                           // load value inside GPIOC_MODER register
  AND   R1, #~(1<<5)                                       // clear the MODER2 bit
  ORR   R1, #(1<<4)                                        // set the MODER2 bit
  STR   R1, [R0]                                           // store value into GPIOC_MODER register
  BX    LR                                                 // return to caller

/**
 * @brief   Configures GPIOC pin 3 to operate in general purpose output mode.
 *
 * @details This function configures GPIOC pin 3 to operate in general purpose output mode.
 *          It modifies the GPIOC_MODER to set the necessary bits for general purpose output mode
 *          on pin 3. The MODER3 bit is set to select general purpose output mode.
 *
 * @param   None
 * @retval  None
 */
GPIOC_PC3_General_Purpose_Output_Mode_Enable:
  LDR   R0, =0x40020800                                    // load address of GPIOC_MODER register
  LDR   R1, [R0]                                           // load value inside GPIOC_MODER register
  AND   R1, #~(1<<7)                                       // clear the MODER3 bit
  ORR   R1, #(1<<6)                                        // set the MODER3 bit
  STR   R1, [R0]                                           // store value into GPIOC_MODER register
  BX    LR                                                 // return to caller

/**
 * @brief   Enables USART1 peripheral and configures its settings for communication.
 *
 * @details This function enables the USART1 peripheral by setting the corresponding
 *          RCC_APB2ENR bit.  It also configures the USART1 settings, including the baud
 *          rate and control register settings.  The USART1_BRR register is set to achieve
 *          a baud rate of 9600, and the USART1_CR1 register is modified to enable USART1
 *          (UE bit) and enable transmission (TE bit).
 *
 * @param   None
 * @retval  None
 */
USART1_Enable:
  LDR   R0, =0x40023844                                    // load address of RCC_APB2ENR register
  LDR   R1, [R0]                                           // load value inside RCC_APB2ENR register
  ORR   R1, #(1<<4)                                        // set the USART1EN bit
  STR   R1, [R0]                                           // store value into RCC_AHB1ENR register
  LDR   R0, =0x40011008                                    // load address of USART1_BRR register
  LDR   R1, [R0]                                           // load value inside USART1_BRR register
  MOV   R1, #0x683                                         // set register to 9600 baud
  STR   R1, [R0]                                           // store value into USART1_BRR register
```

```
  LDR   R0, =0x4001100C                                    // load address of USART1_CR1 register
  LDR   R1, [R0]                                           // load value inside USART1_CR1 register
  ORR   R1, #(1<<13)                                       // set the UE bit
  ORR   R1, #(1<<3)                                        // set the TE bit
  ORR   R1, #(1<<2)                                        // set the RE bit
  STR   R1, [R0]                                           // store value into USART1_CR1 register
  BX    LR                                                 // return to caller

/**
 * @brief   Sends a single character over USART1.
 *
 * @details This function sends a single character over USART1 by writing it to the USART1_DR
 *          register.  It first checks if the transmit buffer is empty (TXE bit) in the
 *          USART1_SR register.  If the buffer is not empty, it waits until it becomes empty
 *          before writing the character to USART1_DR.
 *
 * @param   R7: The character to be sent over USART1.
 * @retval  None
 */
USART1_Transmit_Character:
  LDR   R1, =0x40011000                                    // load address of USART1_SR register
.USART1_Transmit_Character_Loop:
  LDR   R2, [R1]                                           // load value inside USART1_SR register
  AND   R2, #(1<<7)                                        // read TXE bit
  CMP   R2, #0x00                                          // test TX FIFO is not full
  BEQ   .USART1_Transmit_Character_Loop                    // branch if equal
  LDR   R1, =0x40011004                                    // load value inside USART1_DR register
  STR   R7, [R1]                                           // store value into USART1_DR register
  BX    LR                                                 // return to caller

/**
 * @brief   Receives a character over USART1.
 *
 * @details This function receives a character over USART1 by reading the USART1_DR register.
 *          It first checks if the receive buffer is not empty (RXNE bit) in the USART1_SR
 *          register.  If the buffer is empty, it waits until it becomes non-empty before
 *          reading the character from the USART1_DR register. The received character is then
 *          returned.
 *
 * @param   None
 * @retval  R7: The received character over USART1.
 */
USART1_Receive_Character:
  LDR   R0, =0x40011000                                    // load address of USART1_SR register
.USART1_Receive_Character_Loop:
  LDR   R1, [R0]                                           // load value inside USART1_SR register
  AND   R1, #(1<<5)                                        // read the RXNE bit
  CMP   R1, #0x00                                          // test TX FIFO is not full
  BEQ   .USART1_Receive_Character_Loop                     // branch if equal
  LDR   R2, =0x40011004                                    // load value inside USART1_DR register
  LDR   R7, [R2]                                           // read value inside USART1_DR register
  BX    LR                                                 // return to caller

/**
 * @brief   Configures GPIO pins for clockwise full drive sequence mode.
 *
 * @details In full drive sequence mode, two coils are energized at a time, providing full torque to
 *          the stepper motor. This function configures the GPIO pins to control a UNL2003 driver
 *          for clockwise full drive sequence mode operation.
 *
 * @param   R7: The millisecond delay value.
 * @retval  None
 */
Clockwise_Rotation_Sequence:
  PUSH  {LR}                                               // push return address onto stack
  LDR   R0, =0x40020814                                    // load address of GPIOC_ODR register
  LDR   R1, [R0]                                           // load value inside GPIOC_ODR register
  MOV   R1, #0x08                                          // set the ODR register
  STR   R1, [R0]                                           // store value into GPIOC_ODR register
  BL    Delay_MS                                           // call function
  LDR   R0, =0x40020814                                    // load address of GPIOC_ODR register
  LDR   R1, [R0]                                           // load value inside GPIOC_ODR register
  MOV   R1, #0x04                                          // set the ODR register
  STR   R1, [R0]                                           // store value into GPIOC_ODR register
  BL    Delay_MS                                           // call function
  LDR   R0, =0x40020814                                    // load address of GPIOC_ODR register
  LDR   R1, [R0]                                           // load value inside GPIOC_ODR register
  MOV   R1, #0x02                                          // set the ODR register
  STR   R1, [R0]                                           // store value into GPIOC_ODR register
  BL    Delay_MS                                           // call function
  LDR   R0, =0x40020814                                    // load address of GPIOC_ODR register
  LDR   R1, [R0]                                           // load value inside GPIOC_ODR register
  MOV   R1, #0x01                                          // set the ODR register
  STR   R1, [R0]                                           // store value into GPIOC_ODR register
  BL    Delay_MS                                           // call function
  POP   {LR}                                               // pop return address from stack
  BX    LR                                                 // return to caller

/**
 * @brief   Configures GPIO pins for counter-clockwise full drive sequence mode.
 *
 * @details In full drive sequence mode, two coils are energized at a time, providing full torque to
 *          the stepper motor. This function configures the GPIO pins to control a UNL2003 driver
```

```
 *          for counter-clockwise full drive sequence mode operation.
 *
 * @param   R7: The millisecond delay value.
 * @retval  None
 */
Counter_Clockwise_Rotation_Sequence:
  PUSH  {LR}
  LDR   R0, =0x40020814                               // load address of GPIOC_ODR register
  LDR   R1, [R0]                                      // load value inside GPIOC_ODR register
  MOV   R1, #0x01                                     // set the ODR register
  STR   R1, [R0]                                      // store value into GPIOC_ODR register
  BL    Delay_MS                                      // call function
  LDR   R0, =0x40020814                               // load address of GPIOC_ODR register
  LDR   R1, [R0]                                      // load value inside GPIOC_ODR register
  MOV   R1, #0x02                                     // set the ODR register
  STR   R1, [R0]                                      // store value into GPIOC_ODR register
  BL    Delay_MS                                      // call function
  LDR   R0, =0x40020814                               // load address of GPIOC_ODR register
  LDR   R1, [R0]                                      // load value inside GPIOC_ODR register
  MOV   R1, #0x04                                     // set the ODR register
  STR   R1, [R0]                                      // store value into GPIOC_ODR register
  BL    Delay_MS                                      // call function
  LDR   R0, =0x40020814                               // load address of GPIOC_ODR register
  LDR   R1, [R0]                                      // load value inside GPIOC_ODR register
  MOV   R1, #0x08                                     // set the ODR register
  STR   R1, [R0]                                      // store value into GPIOC_ODR register
  BL    Delay_MS                                      // call function
  POP   {LR}                                          // pop return address from stack
  BX    LR                                            // return to caller

/**
 * @brief   Initializes GPIOC PC13 for EXTI interrupt.
 *
 * @details This function configures GPIOC PC13 for EXTI interrupt. It sets the pin's mode
 *          to input and enables the internal pull-up resistor. Additionally, it enables the
 *          EXTI interrupt for PC13, configures SYSCFG_EXTICR4, and sets the corresponding
 *          EXTI and NVIC settings to enable interrupt handling for PC13.
 *
 * @param   None
 * @retval  None
 */
GPIOC_PC13_EXTI_Init:
  PUSH  {LR}                                          // push return address onto stack
  CPSID I                                             // disable global interrupts
  LDR   R0, =0x40020800                               // load address of GPIOC_MODER register
  LDR   R1, [R0]                                      // load value inside GPIOC_MODER register
  AND   R1, #~(1<<27)                                 // clear the MODER13 bit
  AND   R1, #~(1<<26)                                 // clear the MODER13 bit
  STR   R1, [R0]                                      // store value into GPIOC_MODER register
  LDR   R0, =0x4002080C                               // load address of GPIOC_PUPDR register
  LDR   R1, [R0]                                      // load value inside GPIOC_PUPDR register
  AND   R1, #~(1<<27)                                 // clear the PUPDR13 bit
  ORR   R1, #(1<<26)                                  // set the PUPDR13 bit
  STR   R1, [R0]                                      // store value into GPIOC_PUPDR register
  LDR   R0, =0x40023844                               // load address of RCC_ABP2ENR
  LDR   R1, [R0]                                      // load value inside RCC_ABP2ENR register
  ORR   R1, #(1<<14)                                  // set SYSCFGEN bit
  STR   R1, [R0]                                      // store value into RCC_APB2ENR register
  LDR   R0, =0x40013814                               // load address of SYSCFG_EXTICR4
  LDR   R1, [R0]                                      // load value inside SYSCFG_EXTICR4 register
  ORR   R1, #(1<<5)                                   // set EXTI13 bit
  STR   R1, [R0]                                      // store value into SYSCFG_EXTICR4 register
  LDR   R0, =0x40013C00                               // load address of EXTI_IMR register
  LDR   R1, [R0]                                      // load value inside EXTI_IMR register
  ORR   R1, #(1<<13)                                  // set MR13 bit
  STR   R1, [R0]                                      // store value into EXTI_IMR register
  LDR   R0, =0x40013C0C                               // load address of EXTI_FTSR register
  LDR   R1, [R0]                                      // load value inside EXTI_FTSR register
  ORR   R1, #(1<<13)                                  // set TR13 bit
  STR   R1, [R0]                                      // store value into EXTI_IMR register
  BL    NVIC_EnableIRQ_EXTI15_10                      // call function
  CPSIE I                                             // enable global interrupts
  POP   {LR}                                          // pop return address from stack
  BX    LR                                            // return to caller

/**
 * @brief   EXTI callback function for centrifuge control.
 *
 * @details This EXTI callback function simulates centrifuge control and communication.
 *          It includes a loop that mimics a sensor reading delay and sends appropriate
 *          characters over USART1 based on the sensor's value. If the sensor value is
 *          within the normal range, it sends "NORMAL," and if it's high, it sends "HIGH."
 *          The function also checks for a kill switch condition and finishes if engaged.
 *
 * @param   None
 * @retval  None
 */
EXTI_Callback:
  PUSH  {LR}                                          // push return address onto stack
.EXTI_Callback_Loop:
  MOV   R7, #0x40                                     // 64 ms delay variable mock sensor read speed, 8 ms would damage centrifuge
  BL    Clockwise_Rotation_Sequence                   // call function
  CMP   R7, #0x40                                     // compare speed to normal value
```

```
  BNE   .EXTI_Callback_Loop_High_Value                       // branch if not equal
  BLE   .EXTI_Callback_Loop_Normal_Value                     // branch if less than or equal
.EXTI_Callback_Loop_Normal_Value:
  MOV   R7, #0x4E                                            // 'N'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x4F                                            // 'O'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x52                                            // 'R'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x4D                                            // 'M'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x41                                            // 'A'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x4C                                            // 'L'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x0D                                            // '\r'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x0A                                            // '\n'
  BL    USART1_Transmit_Character                            // call function
  BL    Kill_Switch                                          // call function
  CMP   R7, #0x01                                            // compare if kill switch was engaged
  BEQ   .EXTI_Callback_Finish                                // branch if equal
  B     .EXTI_Callback_Loop                                  // unconditional branch
.EXTI_Callback_Loop_High_Value:
  MOV   R7, #0x48                                            // 'H'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x49                                            // 'I'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x47                                            // 'G'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x48                                            // 'H'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x0D                                            // '\r'
  BL    USART1_Transmit_Character                            // call function
  MOV   R7, #0x0A                                            // '\n'
  BL    USART1_Transmit_Character                            // call function
  BL    Kill_Switch                                          // call function
  CMP   R7, #0x01                                            // compare if kill switch was engaged
  BEQ   .EXTI_Callback_Finish                                // branch if equal
  B     .EXTI_Callback_Loop                                  // unconditional branch
.EXTI_Callback_Finish:
  POP   {LR}                                                 // pop return address from stack
  BX    LR                                                   // return to caller

/**
 * @brief   Kill switch handler function.
 *
 * @details This function checks for a specific character received over USART1, and if it
 *          matches '1', it sets a return value in R7 to indicate that the centrifuge should
 *          be stopped. This function is used to implement a kill switch functionality to
 *          stop the centrifuge from spinning.
 *
 * @param   None
 * @retval  R7: A return value indicating whether the centrifuge should be stopped (0x01) or not.
 */
Kill_Switch:
  LDR   R0, =0x40011004                                      // load value inside USART1_DR register
  LDR   R0, [R0]                                             // read value inside USART1_DR register
  CMP   R0, #0x31                                            // compare received character with '1'
  BNE   .Kill_Switch_Finish                                  // branch if not equal
  MOV  R7, #0x01                                             // return value to kill the centrifuge from spinning
.Kill_Switch_Finish:
  BX    LR                                                   // return to caller

/**
 * @brief   Delay function in milliseconds.
 *
 * @details This function provides a software-based delay in milliseconds. It takes an
 *          argument in R7, representing the number of milliseconds to delay. The function
 *          uses nested loops to create the delay, where the inner loop accounts for the
 *          approximate execution time of 1 millisecond at a 16 MHz clock frequency.
 *
 * @param   R7: The number of milliseconds to delay.
 * @retval  None
 */
Delay_MS:
  PUSH  {R7}                                                 // store ms variable
  MOV   R1, #0x00                                            // initialize R1 to 0
.Delay_MS_Outer_Loop:
  CMP   R7, #0x00                                            // compare R7 to 0x00
  BLE   .Delay_MS_Exit                                       // branch if less than or equal to
  MOV   R2, #0xA28                                           // move 1 ms value (at 16 MHz clock) into R2
.Delay_MS_Inner_Loop:
  SUBS  R2, R2, #0x01                                        // decrement the inner loop counter
  BNE   .Delay_MS_Inner_Loop                                 // branch if not equal
  SUBS  R7, R7, #0x01                                        // decrement the outer loop counter
  B     .Delay_MS_Outer_Loop                                 // branch
.Delay_MS_Exit:
  POP   {R7}                                                 // restore ms variable
  BX    LR                                                   // return to caller

/**
 * @brief   Enable NVIC (Nested Vectored Interrupt Controller) for EXTI15_10 interrupts.
```

```
 *
 * @details This function enables the NVIC for EXTI15_10 interrupts. It specifically targets
 *          the NVIC_ISER1 register, which controls interrupts 32 to 63, and sets the bit
 *          corresponding to EXTI15_10 (bit 8) to enable the interrupt handling for EXTI lines
 *          15 to 10.
 *
 * @param   None
 * @retval  None
 */
NVIC_EnableIRQ_EXTI15_10:
  LDR    R0, =0xE000E104                              // NVIC_ISER1, p 683 M7 Arch ref manual, ISER1 interrupts 32-63
  LDR    R1, [R0]                                     // load value inside NVIC_ISER1 register
  ORR    R1, #(1<<8)                                  // 0x100=EXTI15-10 (p 204 Ref Manual), p 210 M4 Programming manual, ISER1 8 is
40
  STR    R1, [R0]                                                         // store value into R0
  BX     LR                                           // return to caller
```

We see now in the __start function instead of putting all the instructions in one place we are logically groping logic that can be reused into separate functions.

In __start, we use the BL or branch long instruction to call each function.

We are familiar with the vector table and what we are going to do now is actually set up an interrupt.

The EXTI15_10 interrupt on the STM32F4 is an external interrupt that can be generated by any of the GPIO pins on lines 10 to 15. This means that any GPIO pin on the STM32F4 can be configured to generate an EXTI15_10 interrupt.

The EXTI15_10 interrupt can be configured to trigger on either a rising edge, falling edge, or both edges of the GPIO signal. It can also be configured to be level-sensitive, which means that the interrupt will be triggered if the GPIO signal is held at a high or low level for a specified period of time.

The EXTI15_10 interrupt can be used to implement a variety of different applications, such as:

Button press detection
Sensor input monitoring
I/O port monitoring
Wakeup from low power modes
Motors ;)

To use the EXTI15_10 interrupt, you need to first configure the GPIO pin to be an interrupt. You can do this using the GPIO peripheral driver. Once the GPIO pin has been configured as an interrupt, you need to enable the EXTI15_10 interrupt in the NVIC.

When the GPIO pin generates an interrupt, the EXTI15_10 interrupt handler will be called. The interrupt handler can then be used to implement the desired application logic.
Here are some examples of how the EXTI15_10 interrupt can be used:

You could use the EXTI15_10 interrupt to detect when a button is pressed. When the button is pressed, the interrupt handler could be used to turn on an LED or start a timer.

You could use the EXTI15_10 interrupt to monitor the state of a sensor. When the sensor detects a change in state, the interrupt handler could be used to take appropriate action, such as logging the data or sending a notification.

You could use the EXTI15_10 interrupt to wake up the microcontroller from low power mode. When the microcontroller wakes up, the interrupt handler could be used to start up the main application.

The EXTI15_10 interrupt is a versatile interrupt that can be used to implement a variety of different applications.

We are going to set up the blue button on the dev board to react when pressed such that we can have running code and the button have it’s own functionality when the interrupt is triggered. We will get into what this will trigger a bit later

On line 116 of our code above, we have to remove the weak reference and add a .word as follows.

```
  .word EXTI15_10_IRQHandler                              // EXTI Line[15:10] interrupts
```

This will allow us to define the interrupt to get triggered.

We then define the actual handler function when this interrupt is triggered.

```
.section .text.EXTI15_10_IRQHandler
.weak EXTI15_10_IRQHandler
.type EXTI15_10_IRQHandler, %function
EXTI15_10_IRQHandler:
  PUSH  {LR}                                              // push return address onto stack
  LDR   R0, =0x40013C14                                   // load the address of EXTI_PR register
  LDR   R1, [R0]                                          // load value inside EXTI_PR register
  TST   R1, #(1<<13)                                      // read the PR13 bit, if 0, then BEQ
  BEQ   .PR13_0                                           // branch if equal
  ORR   R1, #(1<<13)                                      // set the PR13 bit
  STR   R1, [R0]                                          // store value inside R1 into R0
  BL    EXTI_Callback                                     // call function
.PR13_0:
  POP   {LR}                                              // pop return address from stack
  BX    LR                                                // return from the function
```

Other than enabling the clocks on GPIOA and GPIOC we have to set up PC13, our blue button.

```
GPIOC_PC13_EXTI_Init:
  PUSH  {LR}                                                  // push return address onto stack
  CPSID I                                                     // disable global interrupts
  LDR   R0, =0x40020800                                       // load address of GPIOC_MODER register
  LDR   R1, [R0]                                              // load value inside GPIOC_MODER register
  AND   R1, #~(1<<27)                                         // clear the MODER13 bit
  AND   R1, #~(1<<26)                                         // clear the MODER13 bit
  STR   R1, [R0]                                              // store value into GPIOC_MODER register
  LDR   R0, =0x4002080C                                       // load address of GPIOC_PUPDR register
  LDR   R1, [R0]                                              // load value inside GPIOC_PUPDR register
  AND   R1, #~(1<<27)                                         // clear the PUPDR13 bit
  ORR   R1, #(1<<26)                                          // set the PUPDR13 bit
  STR   R1, [R0]                                              // store value into GPIOC_PUPDR register
  LDR   R0, =0x40023844                                       // load address of RCC_ABP2ENR
  LDR   R1, [R0]                                              // load value inside RCC_ABP2ENR register
  ORR   R1, #(1<<14)                                          // set SYSCFGEN bit
  STR   R1, [R0]                                              // store value into RCC_APB2ENR register
  LDR   R0, =0x40013814                                       // load address of SYSCFG_EXTICR4
  LDR   R1, [R0]                                              // load value inside SYSCFG_EXTICR4 register
  ORR   R1, #(1<<5)                                           // set EXTI13 bit
  STR   R1, [R0]                                              // store value into SYSCFG_EXTICR4 register
  LDR   R0, =0x40013C00                                       // load address of EXTI_IMR register
  LDR   R1, [R0]                                              // load value inside EXTI_IMR register
  ORR   R1, #(1<<13)                                          // set MR13 bit
  STR   R1, [R0]                                              // store value into EXTI_IMR register
  LDR   R0, =0x40013C0C                                       // load address of EXTI_FTSR register
  LDR   R1, [R0]                                              // load value inside EXTI_FTSR register
  ORR   R1, #(1<<13)                                          // set TR13 bit
  STR   R1, [R0]                                              // store value into EXTI_IMR register
  BL    NVIC_EnableIRQ_EXTI15_10                              // call function
  CPSIE I                                                     // enable global interrupts
  POP   {LR}                                                  // pop return address from stack
  BX    LR                                                    // return to caller
```

The EXTICR4 register maps GPIO pins to EXTI lines. The EXTI lines are used to generate external interrupts. The EXTICR4 register specifies which GPIO pin is connected to which EXTI line.

The EXTI_IMR register is the Interrupt Mask Register. It controls which EXTI lines are enabled to generate interrupts. If a bit in the EXTI_IMR register is set to 1, then the corresponding EXTI line is enabled to generate interrupts.

The EXTI_FTSR register is the Falling Trigger Selection Register. It specifies which EXTI lines are triggered by falling edges of the GPIO signal. If a bit in the EXTI_FTSR register is set to 1, then the corresponding EXTI line is triggered by falling edges of the GPIO signal.

The following is a detailed explanation of how the code snippet above works:

The first few lines of code disable global interrupts. This is necessary because the following code will be modifying the GPIO and EXTI registers, which can be affected by interrupts.

The next few lines of code configure GPIOC pin 13 as an input pin with pull-up enabled.

The next line of code enables the SYSCFG clock. The SYSCFG peripheral is used to map GPIO pins to EXTI lines.

The next line of code sets the EXTI13 bit in the SYSCFG_EXTICR4 register. This maps GPIOC pin 13 to EXTI line 13.

The next two lines of code enable the EXTI13 interrupt in the EXTI_IMR and EXTI_FTSR registers. This means that the EXTI13 interrupt will be generated when the GPIOC pin 13 signal goes from high to low or low to high.

The final line of code enables the EXTI15_10 interrupt in the NVIC. This means that the EXTI15_10 interrupt handler will be called when the EXTI13 interrupt is generated.

Here is a summary of how the EXTICR4, EXTI_IMR, and EXTI_FTSR registers work together:

The EXTICR4 register maps GPIO pins to EXTI lines. The EXTI_IMR register controls which EXTI lines are enabled to generate interrupts. The EXTI_FTSR register specifies which EXTI lines are triggered by falling edges of the GPIO signal.

By configuring the EXTICR4, EXTI_IMR, and EXTI_FTSR registers, you can configure which GPIO pins generate which EXTI interrupts and how the EXTI interrupts are triggered.

The next item we will cover is functions.

A function in assembly language is a block of code that performs a specific task. Functions can be called from other functions, and they can return values to the caller.

To create a function in assembly language, you need to define a label for the function. The label should be followed by a colon (:). The code for the function should be placed below the label. You will see this everywhere in our prior and current code.

To call a function, you need to use the BL instruction. The BL instruction will push the return address onto the stack and then jump to the function.

To return from a function, you need to use the BX LR instruction. The BX LR instruction will pop the return address from the stack and then jump to the address specified by the return address.

PUSH {LR} and POP {LR}

The PUSH {LR} instruction pushes the link register onto the stack. The link register is used to store the return address of the current function.

The POP {LR} instruction pops the link register from the stack.

You should use the PUSH {LR} and POP {LR} instructions if your function calls other functions. This is because the link register will be overwritten by the return address of the called function.

However, if your function does not call any other functions, then you can simply use the BX LR instruction to return from the function.

Next we will discuss USART and UART.

USART and UART are commonly used terms in the context of serial communication. Let me explain what USART and UART are and how USART1 works on the STM32F401 microcontroller.

UART (Universal Asynchronous Receiver/Transmitter):

UART is a widely used serial communication protocol. It's a hardware module that facilitates the serial transmission and reception of data. UART communication is asynchronous, which means data is sent without a shared clock signal between the sender and receiver. Instead, both sides agree on a baud rate, which determines the data transmission speed.

Key features of UART:

Asynchronous Communication: UART communication does not require a shared clock signal. Instead, it uses start and stop bits to frame each data byte.

Full Duplex: UART allows for full-duplex communication, meaning data can be transmitted and received simultaneously.

Configurable Baud Rate: Baud rate is adjustable to control the data transfer speed.

Widely Used: UART is a simple and widely supported protocol, making it suitable for various applications.

USART (Universal Synchronous Asynchronous Receiver/Transmitter):

USART is an enhanced version of UART that adds the option for synchronous communication in addition to asynchronous. It provides greater flexibility by allowing both synchronous and asynchronous modes of communication. In synchronous mode, a clock signal is used for precise timing, while asynchronous mode follows the UART principles.

Key features of USART:

Synchronous and Asynchronous Modes: USART supports both synchronous and asynchronous communication. In synchronous mode, both sender and receiver share a clock signal for precise timing.

Full Duplex: Like UART, USART allows full-duplex communication.

Configurable Baud Rate: Baud rate is adjustable, similar to UART.

Now, let's focus on USART1 on the STM32F401 microcontroller:

USART1 on STM32F401:

USART1 is one of the USART peripherals available on the STM32F401 microcontroller. It can be configured to work in both asynchronous (UART-like) and synchronous modes, depending on your application's requirements.

Here's a simplified overview of how USART1 works:

Configuration: To use USART1, you configure it by setting various registers like Baud Rate Register (USART_BRR), Control Register 1 (USART_CR1), Control Register 2 (USART_CR2), etc. These registers control aspects like the baud rate, data frame format, and enabling or disabling the USART.

Transmitting Data: To send data via USART1, you load data into the Transmit Data Register (USART_DR). The USART1 hardware takes care of sending the data out serially.

Receiving Data: When receiving data, USART1 stores the received data in the Receive Data Register (USART_DR). You can read this register to retrieve the received data.

Interrupts and DMA: STM32 microcontrollers provide options to use interrupts or DMA (Direct Memory Access) for more efficient data transmission and reception with USART1.

Error Handling: USART1 also provides error flags and mechanisms to handle errors such as framing errors, parity errors, and noise errors.

Clocking: In synchronous mode, you need to configure the clock source and polarity for synchronization.

In summary, USART1 on the STM32F401 microcontroller is a versatile communication peripheral that can be used for both asynchronous (UART) and synchronous communication. It is highly configurable and provides various features for transmitting and receiving data in full-duplex mode. Configuration and control are achieved by setting specific registers provided by the microcontroller's USART1 hardware.

We have functions to enable GPIOC where our pins will exist as well as set them in an alternate function mode.

Alternate Function mode, often referred to as AF mode or Alternate Functionality, is a feature found in many microcontrollers, including those in the STM32 series (such as STM32F401) and other families like the ARM Cortex-M based chips. It allows GPIO (General-Purpose Input/Output) pins to take on roles other than their default digital input/output functions, expanding the versatility of these pins.

Here's a more detailed description of Alternate Function mode and how it works:

Default GPIO Mode: Each GPIO pin on a microcontroller typically has a default digital input/output function. For example, a GPIO pin may be configured by default as a general-purpose digital input or output.

Alternate Functionality: In addition to their default functions, GPIO pins can often be configured to serve other purposes, such as analog input, USART (serial communication), PWM (Pulse Width Modulation), SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), or other specialized functions.

Peripheral Selection: To enable alternate functionality for a GPIO pin, you need to configure the pin's Alternate Function Register (AFR) or a similar configuration register. This register allows you to select the specific peripheral or alternate function that you want the pin to serve.

Pin Multiplexing: Many microcontrollers have multiple peripherals that can use the same GPIO pins for different purposes. Alternate Function mode enables pin multiplexing, where the same physical pin can be shared between multiple peripherals. You select which peripheral to use by configuring the appropriate AFR bits for the pin.

Control and Configuration: Besides selecting the peripheral, you can often configure various parameters related to the peripheral function. For example, you can set the pin as an input or output, configure the pin as open-drain or push-pull, and set the pin's pull-up or pull-down resistors, among other settings.

Flexibility: Alternate Function mode provides flexibility in designing embedded systems. It allows you to optimize pin usage and resource allocation on your microcontroller, making it possible to interface with a wide range of external devices and communication protocols.

Example: For instance, if you want to use a GPIO pin for UART communication (USART), you can configure that pin in Alternate Function mode and select the USART peripheral associated with it. This way, the same pin can serve as a digital GPIO pin or as a USART transmit or receive pin, depending on the configuration.

Documentation: To use Alternate Function mode effectively, consult your microcontroller's datasheet and reference manual, which provide detailed information about the available alternate functions for each GPIO pin and how to configure them.

In summary, Alternate Function mode is a powerful feature in microcontrollers that allows you to assign various peripheral functions to GPIO pins, expanding the functionality and versatility of your embedded applications. It enables pin multiplexing, allowing multiple peripherals to share the same physical pins and giving you greater flexibility in designing your hardware interfaces.

Let’s briefly discuss the industrial control hack which was STUXNET.

STUXNET was a highly sophisticated malware that was designed to target industrial control systems (ICS). It is believed to have been developed by a joint operation between the United States and Israel, and was used to attack Iran's nuclear program in 2010.

STUXNET is a complex piece of malware that uses a variety of techniques to evade detection and achieve its goals. It is also highly modular, which makes it difficult to analyze and understand.

STUXNET is a multi-stage malware that consists of a number of different components. The following is a brief overview of the technical details of STUXNET:

Stage 1: STUXNET is initially delivered to the target system as a malicious USB drive. The USB drive contains a malicious file that is disguised as a legitimate software update.

Stage 2: When the malicious file is executed, it installs a rootkit on the target system. The rootkit gives STUXNET full control of the target system and allows it to evade detection.

Stage 3: STUXNET then searches for specific industrial control systems on the target network. When STUXNET finds a target ICS, it installs a malicious firmware update on the ICS.

Stage 4: The malicious firmware update modifies the ICS to behave in a way that is favorable to the attacker. For example, STUXNET can be used to cause the ICS to spin centrifuges at too high a speed, which can damage the centrifuges.

STUXNET also uses a number of other techniques to evade detection and achieve its goals. For example, STUXNET uses a variety of encryption techniques to protect its code and data. STUXNET also uses a technique called code obfuscation to make its code difficult to analyze.

In sum, STUXNET was a highly sophisticated malware that is designed to target industrial control systems. It is a complex piece of malware that uses a variety of techniques to evade detection and achieve its goals. STUXNET is a significant threat to industrial control systems, and organizations need to take steps to protect themselves from STUXNET and other similar malware.

Here are some additional technical details about STUXNET:

STUXNET uses four zero-day vulnerabilities to exploit Windows systems.

STUXNET uses a custom rootkit to hide its presence on the target system.
STUXNET targets specific industrial control systems, such as Siemens WinCC/SCADA systems.

STUXNET modifies the firmware of industrial control systems to cause them to behave in a way that is favorable to the attacker.

STUXNET uses a variety of encryption techniques to protect its code and data.

STUXNET uses code obfuscation to make its code difficult to analyze.

STUXNET is a significant threat to industrial control systems, and organizations need to take steps to protect themselves from STUXNET and other similar malware. Organizations can protect themselves from STUXNET by implementing the following measures:

Keeping Windows systems up to date with the latest security patches.

Using a firewall to protect networks from unauthorized access.

Using intrusion detection and prevention systems to monitor networks for suspicious activity.

Educating employees about the dangers of malware and how to protect themselves.

Organizations should also consider implementing specific measures to protect their industrial control systems, such as:

Using air gaps to isolate industrial control systems from the internet.

Using security solutions that are specifically designed for industrial control systems.

Implementing regular security assessments of industrial control systems.

Let’s put it all together!

We will create a STUXNET style demo where we will use the motor and controller with our board.

Instead of a SIEMENS PLC we will be using our MCU to drive the motor at a normal speed.

We need to snap in the white connector on the motor to the ULN2003 as it can only go in one way. We then need to connect PC0(MCU) to IN1(ULN2003), PC1(MCU) to IN2(ULN2003), PC2(MCU) to IN3(ULN2003) and PC4(MCU) to IN4(ULN2003). Finally we connect the 5V(MCU) to 5V(ULN2003).

We are going to program up the demo with a button interrupt where when pressed it will start the motor and spin the mock centrifuge at a normal speed.

We will have our UART connected to a HM11 or an FTDI connector to be plugged into a USB device.

If you use the HM11, you will use 3.3V power(MCU) to 3.3V(HM11) and GND(MCU) to GND(HM11) and the TX(MCU) to RX(HM11) and RX(MCU) to TX(HM11).

If you are using the FTDI connector USE ONLY THE WHITE AND GREEN WIRES AS YOU DO NOT WANT TO CONNECT POWER AS THE BOARD IS ALREADY POWERED UP! You would connect TX WHITE(FTDI) to PA9(MCU) and RX Green(FTDI) to PA10(MCU)

Let's see this in action by first assembling then linking and finally flashing to our MCU.

The difference here is we are not going to keep symbols we are going to examine this firmware as one would investigate in the wild so this will be significantly more challenging.

We also need to download the STM32CubeProgrammer to patch our binary in real-time once we find our areas of attack.

https://www.st.com/en/development-tools/stm32cubeprog.html#get-software

Our situation is we have been contracted by a classified organization to infiltrate the Natanz Nuclear Facility. The intel we have been given is that a normal rate of delay is 64 ms for the centrifuge to spin from a classified source and that if it ever reached 8 it would spin the centrifuge out of control and destroy the facility which is our goal ;) The intel provided also explains that by entering in a 1 into the terminal will act as a kill switch in the event that an Operator obverses something wrong.

The other piece of intel is that the motor is controlled by a UNL2003 driver with a stepper motor and it is designed in a full drive sequence mode which two coils are energized at a time that means two windings of stepper motor energized together. Therefore, motor runs at full torque.

Let's assemble...

```
arm-none-eabi-as -g main.s -o main.o
```

```
arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld

arm-none-eabi-objcopy -O binary --strip-all main.elf main.bin
openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.bin 0x08000000 verify reset exit"
```

When we fire up a UART and press the blue button we see the motor start to spin a normal speed and we notice our UART is displaying the following.

```
NORMAL
NORMAL
NORMAL
NORMAL
NORMAL
```

Let’s debug!

Terminal 1:

```
openocd -f board/st_nucleo_f4.cfg
```

Terminal 2:

```
arm-none-eabi-gdb main.bin

target remote :3333

monitor reset halt
```

Now that we are inside the firmware, we have to find the entry point as we do not have symbols. Let’s look at the first 1000 instructions.

```
(gdb) x/1000i 0x08000000
```

We see a good deal of startup code which is not particularly of value however after a few pages we find BL to a number of functions, this is of interest to us.

```
   0x80001c0:   bl      0x80001ea
   0x80001c4:   bl      0x80001f6
   0x80001c8:   bl      0x8000202
   0x80001cc:   bl      0x8000228
   0x80001d0:   bl      0x800024e
   0x80001d4:   bl      0x800025e
   0x80001d8:   bl      0x800026e
   0x80001dc:   bl      0x800027e
   0x80001e0:   bl      0x800028e
   0x80001e4:   bl      0x8000358
   0x80001e8:   b.n     0x80001e8
```

Let’s take note of these functions for now.

What we need to do is find where our interrupt is triggering. Luckily we can research the Arch Ref Manual. Let’s go to page 682, (document included in GitHub repo).

We see NVIC_ISER0-NVIC_ISER15 so we see the address starting at 0xE000E100 to 0xE000E13C. So lets search our binary for each as they are 4-bytes long.

```
(gdb) find 0x08000000, 0x0A000000, 0xe000e100
Pattern not found.
```

Ok let’s try the next one.

```
(gdb) find 0x08000000, 0x0A000000, 0xe000e104
0x80004b8
1 pattern found.
```

Great! Let’s examine what is at the address minus a few bytes.

```
(gdb) x/100i 0x8000400
   0x8000400:   cmp     r7, #1
   0x8000402:   beq.n   0x8000440
   0x8000404:   b.n     0x80003ae
   0x8000406:   mov.w   r7, #72 ; 0x48
   0x800040a:   bl      0x80002b4
   0x800040e:   mov.w   r7, #73 ; 0x49
   0x8000412:   bl      0x80002b4
   0x8000416:   mov.w   r7, #71 ; 0x47
   0x800041a:   bl      0x80002b4
   0x800041e:   mov.w   r7, #72 ; 0x48
   0x8000422:   bl      0x80002b4
   0x8000426:   mov.w   r7, #13
   0x800042a:   bl      0x80002b4
   0x800042e:   mov.w   r7, #10
   0x8000432:   bl      0x80002b4
   0x8000436:   bl      0x8000446
   0x800043a:   cmp     r7, #1
   0x800043c:   beq.n   0x8000440
   0x800043e:   b.n     0x80003ae
   0x8000440:   ldr.w   lr, [sp], #4
   0x8000444:   bx      lr
   0x8000446:   ldr     r0, [pc, #88]   ; (0x80004a0)
   0x8000448:   ldr     r0, [r0, #0]
   0x800044a:   cmp     r0, #49 ; 0x31
   0x800044c:   bne.n   0x8000452
   0x800044e:   mov.w   r7, #1
   0x8000452:   bx      lr
   0x8000454:   push    {r7}
   0x8000456:   mov.w   r1, #0
--Type <RET> for more, q to quit, c to continue without paging--
   0x800045a:   cmp     r7, #0
   0x800045c:   ble.n   0x800046a
   0x800045e:   movw    r2, #2600       ; 0xa28
   0x8000462:   subs    r2, #1
   0x8000464:   bne.n   0x8000462
   0x8000466:   subs    r7, #1
   0x8000468:   b.n     0x800045a
   0x800046a:   pop     {r7}
   0x800046c:   bx      lr
   0x800046e:   ldr     r0, [pc, #72]   ; (0x80004b8)
   0x8000470:   ldr     r1, [r0, #0]
   0x8000472:   orr.w   r1, r1, #256    ; 0x100
   0x8000476:   str     r1, [r0, #0]
   0x8000478:   bx      lr
   0x800047a:   movs    r0, r0
   0x800047c:   subs    r4, #20
   0x800047e:   ands    r1, r0
   0x8000480:   subs    r0, #48 ; 0x30
   0x8000482:   ands    r2, r0
   0x8000484:   movs    r0, r0
   0x8000486:   ands    r2, r0
   0x8000488:   movs    r4, r4
   0x800048a:   ands    r2, r0
   0x800048c:   lsrs    r0, r0, #32
   0x800048e:   ands    r2, r0
   0x8000490:   subs    r0, #68 ; 0x44
   0x8000492:   ands    r2, r0
   0x8000494:   asrs    r0, r1, #32
   0x8000496:   ands    r1, r0
--Type <RET> for more, q to quit, c to continue without paging--
   0x8000498:   asrs    r4, r1, #32
   0x800049a:   ands    r1, r0
   0x800049c:   asrs    r0, r0, #32
   0x800049e:   ands    r1, r0
   0x80004a0:   asrs    r4, r0, #32
   0x80004a2:   ands    r1, r0
   0x80004a4:   lsrs    r4, r2, #32
```

```
   0x80004a6:   ands    r2, r0
   0x80004a8:   lsrs    r4, r1, #32
   0x80004aa:   ands    r2, r0
   0x80004ac:   subs    r0, #20
   0x80004ae:   ands    r1, r0
   0x80004b0:   subs    r4, #0
   0x80004b2:   ands    r1, r0
   0x80004b4:   subs    r4, #12
   0x80004b6:   ands    r1, r0
   0x80004b8:   b.n     0x80006c4
   0x80004ba:   b.n     0x80004be
```

Ok there can be something here. One thing I notice is 0x48, 0x49, 0x47, 0x48, 13 and 10 and I wonder can these be ascii chars going to the UART? The 13 followed by a 10 is a dead giveaway as that is /r/n respectively.

These spell out HIGH. So this is of interest for sure however we only see NORMAL echoing in our UART so far. We would not want to ever see HIGH as that would let the victim know things are not normal so this is something to keep in mind.

We know from our intel that a 1 or 0x31 will be compared somewhere in the code as we will need to hack this to be a non-printable compare like 0x01 or something like that. If we re-examine our code we find just that!

```
0x800044a:   cmp     r0, #49 ; 0x31
```

Let's make note of this when we patch our firmware.

Next we know that 64ms is the value we must find in order to change this to 8ms.

We don't see these values but lets examine farther back in our code and see if something pops out.

```
(gdb) x/100i 0x8000300
   0x8000300:   bl      0x8000454
   0x8000304:   ldr     r0, [pc, #412]  ; (0x80004a4)
   0x8000306:   ldr     r1, [r0, #0]
   0x8000308:   mov.w   r1, #1
   0x800030c:   str     r1, [r0, #0]
   0x800030e:   bl      0x8000454
   0x8000312:   ldr.w   lr, [sp], #4
   0x8000316:   bx      lr
   0x8000318:   push    {lr}
   0x800031a:   ldr     r0, [pc, #392]  ; (0x80004a4)
   0x800031c:   ldr     r1, [r0, #0]
   0x800031e:   mov.w   r1, #1
   0x8000322:   str     r1, [r0, #0]
   0x8000324:   bl      0x8000454
   0x8000328:   ldr     r0, [pc, #376]  ; (0x80004a4)
   0x800032a:   ldr     r1, [r0, #0]
   0x800032c:   mov.w   r1, #2
   0x8000330:   str     r1, [r0, #0]
   0x8000332:   bl      0x8000454
   0x8000336:   ldr     r0, [pc, #364]  ; (0x80004a4)
   0x8000338:   ldr     r1, [r0, #0]
   0x800033a:   mov.w   r1, #4
   0x800033e:   str     r1, [r0, #0]
   0x8000340:   bl      0x8000454
   0x8000344:   ldr     r0, [pc, #348]  ; (0x80004a4)
   0x8000346:   ldr     r1, [r0, #0]
   0x8000348:   mov.w   r1, #8
   0x800034c:   str     r1, [r0, #0]
   0x800034e:   bl      0x8000454
--Type <RET> for more, q to quit, c to continue without paging--
   0x8000352:   ldr.w   lr, [sp], #4
   0x8000356:   bx      lr
   0x8000358:   push    {lr}
```

```
   0x800035a:   cpsid   i
   0x800035c:   ldr     r0, [pc, #300]  ; (0x800048c)
   0x800035e:   ldr     r1, [r0, #0]
   0x8000360:   bic.w   r1, r1, #134217728      ; 0x8000000
   0x8000364:   bic.w   r1, r1, #67108864       ; 0x4000000
   0x8000368:   str     r1, [r0, #0]
   0x800036a:   ldr     r0, [pc, #316]  ; (0x80004a8)
   0x800036c:   ldr     r1, [r0, #0]
   0x800036e:   bic.w   r1, r1, #134217728      ; 0x8000000
   0x8000372:   orr.w   r1, r1, #67108864       ; 0x4000000
   0x8000376:   str     r1, [r0, #0]
   0x8000378:   ldr     r0, [pc, #276]  ; (0x8000490)
   0x800037a:   ldr     r1, [r0, #0]
   0x800037c:   orr.w   r1, r1, #16384  ; 0x4000
   0x8000380:   str     r1, [r0, #0]
   0x8000382:   ldr     r0, [pc, #296]  ; (0x80004ac)
   0x8000384:   ldr     r1, [r0, #0]
   0x8000386:   orr.w   r1, r1, #32
   0x800038a:   str     r1, [r0, #0]
   0x800038c:   ldr     r0, [pc, #288]  ; (0x80004b0)
   0x800038e:   ldr     r1, [r0, #0]
   0x8000390:   orr.w   r1, r1, #8192   ; 0x2000
   0x8000394:   str     r1, [r0, #0]
   0x8000396:   ldr     r0, [pc, #284]  ; (0x80004b4)
   0x8000398:   ldr     r1, [r0, #0]
   0x800039a:   orr.w   r1, r1, #8192   ; 0x2000
--Type <RET> for more, q to quit, c to continue without paging--
   0x800039e:   str     r1, [r0, #0]
   0x80003a0:   bl      0x800046e
   0x80003a4:   cpsie   i
   0x80003a6:   ldr.w   lr, [sp], #4
   0x80003aa:   bx      lr
   0x80003ac:   push    {lr}
   0x80003ae:   mov.w   r7, #64 ; 0x40
   0x80003b2:   bl      0x80002d8
   0x80003b6:   cmp     r7, #64 ; 0x40
   0x80003b8:   bne.n   0x8000406
   0x80003ba:   ble.n   0x80003bc
   0x80003bc:   mov.w   r7, #78 ; 0x4e
   0x80003c0:   bl      0x80002b4
   0x80003c4:   mov.w   r7, #79 ; 0x4f
   0x80003c8:   bl      0x80002b4
   0x80003cc:   mov.w   r7, #82 ; 0x52
   0x80003d0:   bl      0x80002b4
   0x80003d4:   mov.w   r7, #77 ; 0x4d
   0x80003d8:   bl      0x80002b4
   0x80003dc:   mov.w   r7, #65 ; 0x41
   0x80003e0:   bl      0x80002b4
   0x80003e4:   mov.w   r7, #76 ; 0x4c
   0x80003e8:   bl      0x80002b4
   0x80003ec:   mov.w   r7, #13
   0x80003f0:   bl      0x80002b4
   0x80003f4:   mov.w   r7, #10
   0x80003f8:   bl      0x80002b4
   0x80003fc:   bl      0x8000446
   0x8000400:   cmp     r7, #1
```

BINGO! We found it! We can stop continuing to page at this point.

```
   0x80003ae:   mov.w   r7, #64 ; 0x40
   0x80003b2:   bl      0x80002d8
   0x80003b6:   cmp     r7, #64 ; 0x40
```

So we can guess that changing that value from 64 or 0x40 to 0x08 will reduce the delay and spin the centrifuge up to a destructive level! We also want to change the compare to 0x08 as well to read normal ;).

We can quit GDB at this point as well as OpenOCD.

Let's fire up the STM32CubeProgrammer and connect to our binary.

After connecting lets seek out the address of 0x0800044a (Address field) which is where the kill switch value is and let's examine 0x4 bytes in the Size field.

```
  0x800044a:   cmp     r0, #49 ; 0x31
```

We see D1012831 so we need to change the last byte to 01.

Let’s disconnect and try it out and we see that when we try to press 1 it no longer stops the centrifuge so we have successfully disabled the kill switch!

Let’s reconnect and find the below.

```
0x80003ae:   mov.w   r7, #64 ; 0x40
0x80003b2:   bl      0x80002d8
0x80003b6:   cmp     r7, #64 ; 0x40
```

Let’s put in the address of 0x080003ae.

We see the value of 0740F04F so we need to change the 40 to 08.

We can disconnect however if we don’t change the compare we will let the Operators know that we are destroying the centrifuge by spinning it up to a dangerous level and we don’t want to do that.

Let’s put in the address of 0x080003b6.

We see the value of D1254F40 so we need to change the 40 to 08.

Now when we fire it up we see the centrifuge spinning out of control (well as fast as this little motor can go LOL) and we also see a NORMAL reading in the UART and that we can’t disable it! SUCCESS!

One final bit of knowledge I can share with you as well is when you come across BLE or BNE (shown as ble.n and bne.n) where BLE is branch if less than or equal and BNE is branch if not equal. If you want to change these codes you will see something like the following.

```
80003c8:       dd00           ble.n   80003cc
80003ca:       d117           bne.n   80003fc
```

Don’t worry about the addresses however focus on the DD00 and D117. If you change DD00 to D100 it will turn the BLE into a BNE and if you change the D117 to DD17 you will reverse the BNE and BLE. You will find more information in the ARMv7-M_Architecture_Reference_Manual located within this repo. Section A7.7.12, Encoding T1, you will find information on the branch instruction. Section A7.3, conditional execution, you will find more info on the binary values for each of the conditions.

You can take time and see what values are in your own code and develop this knowledge for any instruction!

This book is has been quite a jam packed amount of knowledge. I encourage you all to keep going and become the best you can be in the Embedded Engineering and Reverse Engineering world!