FIRST EDITION – CHAPTER 4 REV 1

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

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

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-test
cd 0x0001-test
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 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
  MOV  SP, R0
  BL   __start

/**
 * @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
  B.N  Default_Handler

/**
 * @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
}
```

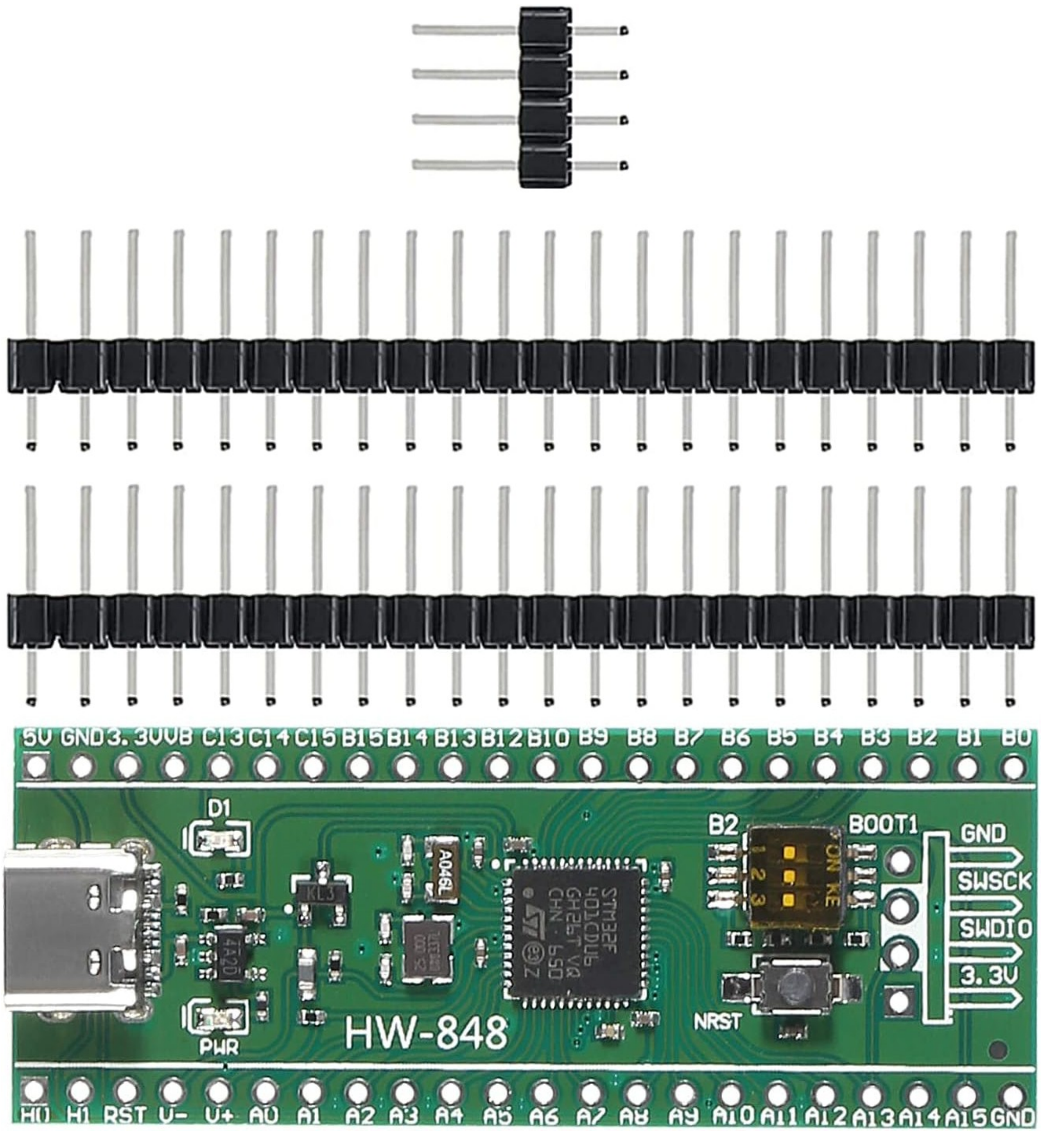
Let’s assemble our simple source code.

```
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.


5V GND 3.3V VB C13 C14 C15 B15 B14 B13 B12 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
D1
B2
BOOT1
GND
SWSCK
SWDIO
3.3V
NRST
PWR
HW-848
H0 H1 RST V- V+ A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 GND

After soldering on 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 an 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.