KEVIN THOMAS

# EMBEDDED REVERSE ENGINEERING

## ARM EDITION

# PREFACE

In today's rapidly evolving technological landscape, embedded devices have become the silent workhorses powering our modern world. From smart home appliances and wearable health monitors to automotive systems and industrial automation, embedded systems are at the heart of countless innovations. Recent estimates suggest there are now over 30 billion embedded devices worldwide, a number projected to soar to 50 billion by 2030. This explosive growth highlights not only the ubiquity of embedded technology but also the immense opportunities and challenges it presents for engineers, developers, and enthusiasts alike.

At the core of most of these devices lies the ARM architecture, renowned for its efficiency, versatility, and widespread adoption across industries. Whether in consumer electronics, medical devices, or critical infrastructure, ARM-based microcontrollers have set the standard for performance and energy efficiency. For anyone aspiring to build, understand, or innovate in the field of embedded systems, a solid grasp of the ARM architecture is essential.

This book is designed to be your companion on a hands-on journey into the world of embedded programming. Using the NUCLEO STM32F401RE development board—a powerful yet accessible platform—we will start from the very basics and progressively explore the intricacies of the ARM architecture. Step by step, you'll learn not just how to write code for embedded systems, but also how to understand the underlying hardware, optimize performance, and develop the mindset of an embedded systems engineer. Whether you are a student, hobbyist, or professional, this book will empower you to unlock the full potential of the ARM architecture and turn your ideas into reality, one line of code at a time.

You will be expected to know C to a moderate level and will need to have VSCode installed with the PlatformIO extension.

You will also need the NUCLEO-F401RE discovery board as well.

With that, let's begin our journey!

# CHAPTER 1: MEMORY MAP

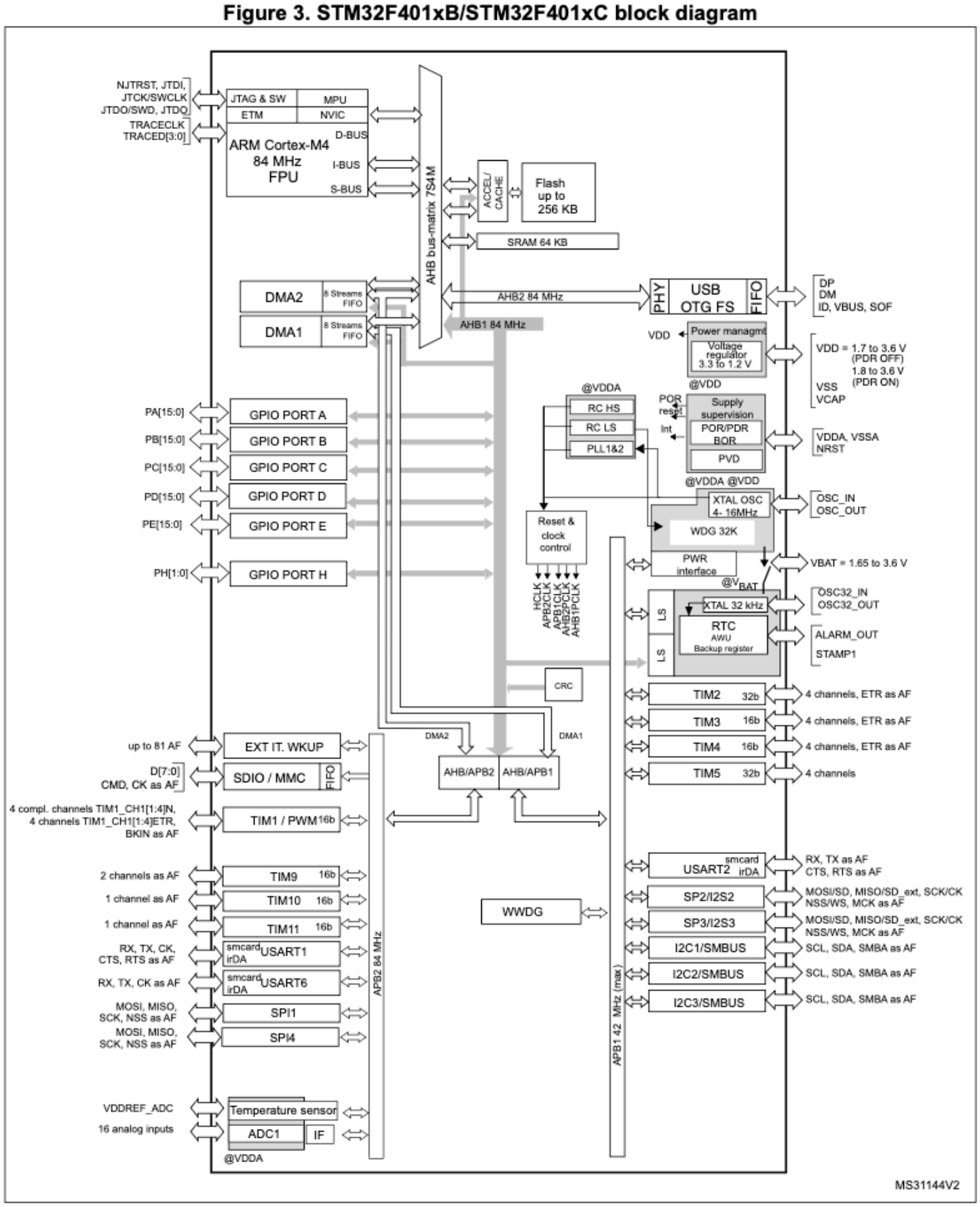
The first thing we are going to do is to understand the memory map of our board. There is a documents folder within this GitHub repo where we will start with the datasheet.

We are dealing with an embedded ARM processor, specifically an STM32F401.

This is a 32-bit processor so the width of the system bus is 32-bits wide. This means there is 2 to the power of 32 possible addressable addresses starting from 0x00000000 to 0xFFFFFFFF.

## BLOCK DIAGRAM

**Figure 3. STM32F401xB/STM32F401xC block diagram**



Let's turn to the block diagram in the datasheet. We see 4 buses to which data will traverse. The first is the APB1 (Advanced Peripheral Bus) which we see in the bottom right of our block diagram. The second is the APB2 bus which we see in the lower left. We see the AHB1 but in the center of the diagram vertically and the AHB2 bus just above that horizontally.

Take note that each of these buses have a number of peripherals tied to them. For example the APB2 bus has USART1 and USART6 as the AHB1 has the GPIO's.

Each of these have physical hardware register addresses that start at 0x40000000.

## MEMORY MAP

Let's turn to the memory map within our reference manual. In 2.3 we find our memory map on page 38 and 39.

If we look at the very bottom we see the APB1 bus's starting address which happens to be the base of the TIM2 peripheral which is 0x40000000.

We can also see that the base of the APB2 bus starts at 0x40010000 with the TIM1 peripheral.

We can continue to derive other peripherals in this table as we will build these out from scratch in C when we begin to build our drivers.

Directly below our table 1 we have our embedded SRAM where on page 42 we see table 4 which maps our embedded SRAM.

We see SRAM1 starts at 0x20000000 and ending at 0x20017FFF and is 96k wide.

We see system memory starting at 0x1FFF0000 and ending at 0x1FFF77FF.

We see our flash memory where our actual program (firmware) exists starting at 0x08000000 and ending at 0x0807FFFF.

## WHERE DATA LIVES

It is important to understand where data lives within our system. Let's look at a few small code snippets.

```
const uint8_t num = 42;
```

This value is constant and will be stored within flash memory somewhere between 0x08000000 and 0x0807FFFF.

```
#include <stdint.h>

#include <string.h>


const char *meaning_of_life = “The number 42!”;

char my_meaning[16];


int main(void) {

    for(uint8_t i = 0; I < strlen(meaning_of_life); i++) {

      my_meaning[i] = *(meaning_of_life + i);

    }


    for(;;);


    return 0;

}
```

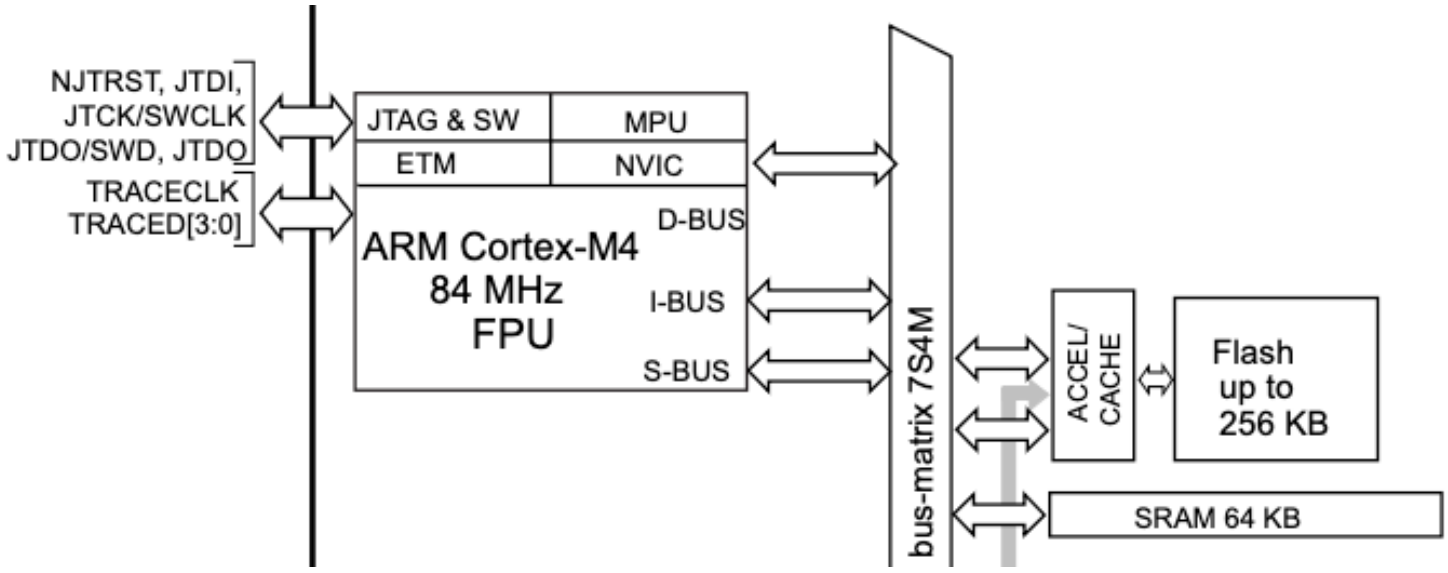
This code snippet starts with a constant pointer that will live in flash memory which is the **meaning_of_life** variable.

We then see the non-constant **my_meaning** variable which will be stored in SRAM.

Finally, we see a data copy from flash to sram with **my_meaning[i] = *(meaning_of_life + i);** where the **meaning_of_life** pointer iterates one byte at a time from flash into the **my_meaning[i]** SRAM.

## HOW THINGS WORK

Let’s revisit our bus matrix and see how things work.




The processor will fetch the instruction on the I-BUS meaning the instructions will be read from flash over the instruction bus.

The processor will use the D-BUS in order to read the data which is stored in flash, therefore it will be read from flash over the data bus.

The instructions of the program will be fetched over the I-BUS and the constant data will be fetched over the D-BUS and in our code snippet, it will then go into SRAM here **my_meaning[i] = *(meaning_of_life + i);** as this all occurs over the AHB bus matrix.

The I-BUS and D-BUS work in parallel.

For more information on the I-BUS and D-BUS you can refer to the ARM Technical Reference Manual included in the documents folder within our GitHub repo where it mentions the ICode memory interface where instructions are fetched from code memory space, 0x00000000 to 0x1FFFFFFC over the 32-bit AHB-Lite bus. In addition, we have the DCode memory interface where data is fetched from code memory space 0x00000000 to 0x1FFFFFFF over the 32-bit AHB-Lite bus.

There is also an S-BUS in the diagram above where in the ARM Technical Reference Manual it mentions that the system interface has instruction fetches and data in addition to debug accesses to address ranges to 0x20000000 to 0xDFFFFFFF and 0xE0100000 to 0xFFFFFFFF which are performed over the 32-bit AHB-Lite bus.

The AHB bus gets converted into two APB buses to which the APB buses are slower. Therefore the peripherals on the APB buses will be slower than the ones on the AHB bus.

# CHAPTER 2: CLOCK

The clock within the MCU literally controls everything and without it, nothing would work.

The MCU is communicates synchronously with digital circuitry through the use of the system clock.

The reference manual, on page 93 has a section 6.2 focusing on clocks. There are three different clock sources used to drive the SYSCLK, system clock.

1. HSI oscillator clock - RC oscillator - internal - default clock, no external parts

2. HSE oscillator clock - crystal oscillator - external - more accurate, requires crystal

3. Main PLL clock - Phased Locked Loop - internal - generates higher frequencies

## HSI CLOCK

The reference manual has on page 97, 6.2.2 HSI clock where it mentions it has a 16 MHz RC oscillator that can be used directly as a system clock, or used as a PLL input.

Upon MCU reset, it will always start with the HSI clock and with code you can change to HSE or PLL.

## HSI CLOCK

The system clock is managed through the RCC registers. On page 103 of the reference manual we have 6.3 RCC registers.

The first thing we need to do is figure out what peripheral we are going to program. Lets start with GPIOA which is a general-purpose input-output series of registers.

We first need to turn to pave 38 of our reference manual and look at table 1 and determine which bus GPIOA is on. We can see it exists on AHB1.

The next step is to turn back to the RCC registers on page 103 of the reference manual and we need to find RCC_AHB1ENR which is 6.3.9 or on page 118.

We see bit 0 handles GPIOAEN which will enable the clock for our desired GPIOA port.

Let’s open VSCode with our **0x01_RCC-clock** example project.

```c
#include <stm32f401xe.h>

int main(void) {
    // Initialize the AHB1 system clock for GPIOA
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // RM p.118

    // Main loop
    while (1) {
    }

    return 0;
}
```

In our next chapter, we will reverse engineer this code.