

KEVIN THOMAS
**EMBEDDED REVERSE
ENGINEERING**
ARM EDITION



Preface	3
Chapter 1: Memory Map	4
Block Diagram	4
Memory Map	5
Where Data Lives.....	7
How Things Work.....	8
Chapter 2: Vector Table	10
Vector Table	10
Flash Memory Placement.....	14
Chapter 3: Boot	16
g_pfnVectors.....	16
__do_global_dtors_aux.....	16
frame_dummy	19
main	22
Chapter 4: Clock	23
HSI Clock	23
HSI Clock	23
Chapter 4: Reversing Clock.....	25
Code Walkthrough.....	25
Code Reversing.....	26

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

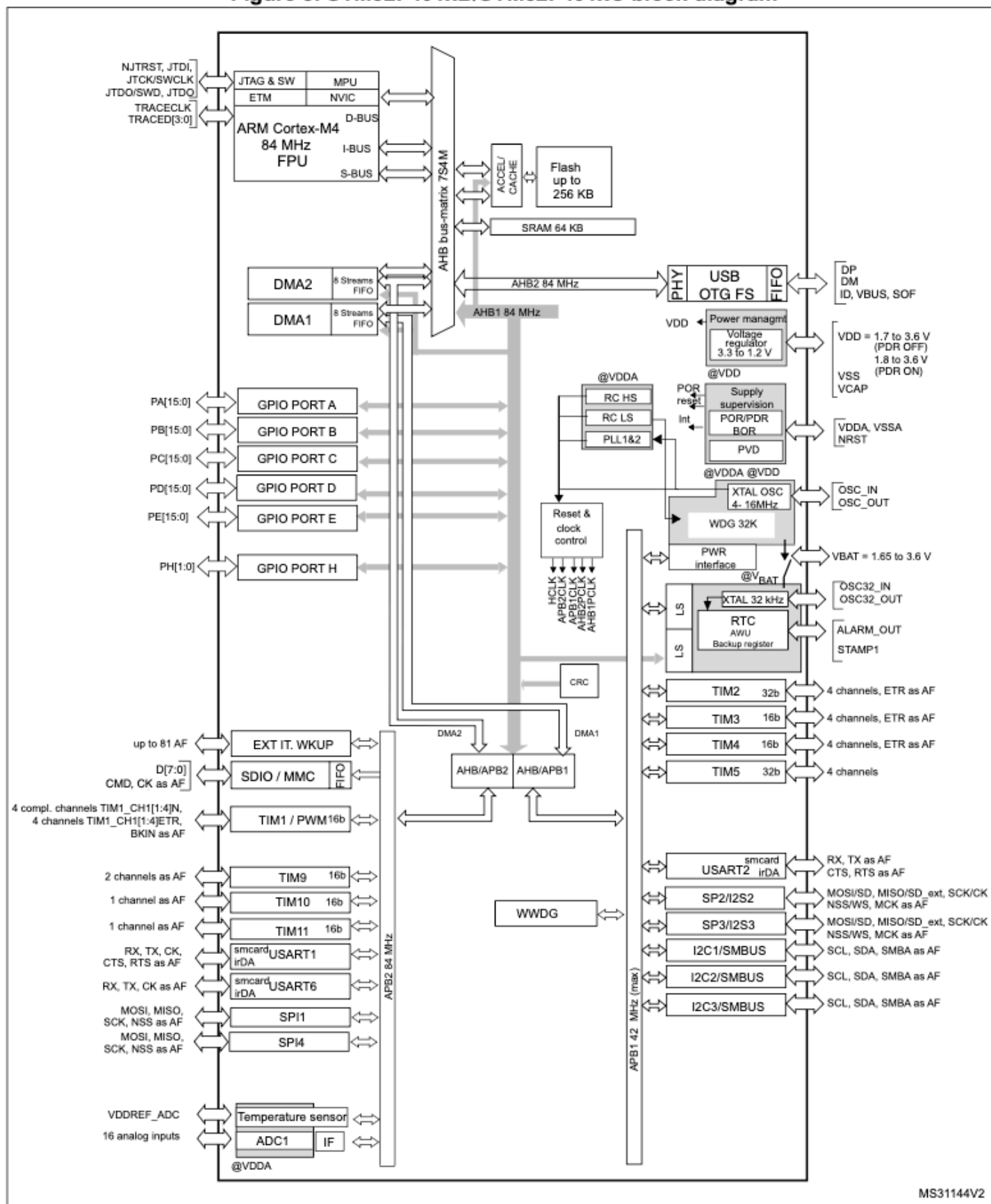
The first thing we are going to do is to understand the memory map of our board. There is a documents folder within this GitHub repo where we will start with the datasheet.

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

Table 1. STM32F401xB/C and STM32F401xD/E register boundary addresses

Boundary address	Peripheral	Bus	Register map
0x5000 0000 - 0x5003 FFFF	USB OTG FS	AHB2	Section 22.16.6: OTG_FS register map on page 755
0x4002 6400 - 0x4002 67FF	DMA2	AHB1	Section 9.5.11: DMA register map on page 198
0x4002 6000 - 0x4002 63FF	DMA1		
0x4002 3C00 - 0x4002 3FFF	Flash interface register		Section 3.8: Flash interface registers on page 60
0x4002 3800 - 0x4002 3BFF	RCC		Section 6.3.22: RCC register map on page 137
0x4002 3000 - 0x4002 33FF	CRC		Section 4.4.4: CRC register map on page 70
0x4002 1C00 - 0x4002 1FFF	GPIOH		Section 8.4.11: GPIO register map on page 164
0x4002 1000 - 0x4002 13FF	GPIOE		
0x4002 0C00 - 0x4002 0FFF	GPIOD		
0x4002 0800 - 0x4002 0BFF	GPIOC		
0x4002 0400 - 0x4002 07FF	GPIOB		
0x4002 0000 - 0x4002 03FF	GPIOA		

Table 1. STM32F401xB/C and STM32F401xD/E register boundary addresses (continued)

Boundary address	Peripheral	Bus	Register map
0x4001 4800 - 0x4001 4BFF	TIM11	APB2	Section 14.5.12: TIM10/11 register map on page 420
0x4001 4400 - 0x4001 47FF	TIM10		
0x4001 4000 - 0x4001 43FF	TIM9		Section 14.4.13: TIM9 register map on page 410
0x4001 3C00 - 0x4001 3FFF	EXTI		Section 10.3.7: EXTI register map on page 212
0x4001 3800 - 0x4001 3BFF	SYSCFG		Section 7.2.8: SYSCFG register map
0x4001 3400 - 0x4001 37FF	SPI4		Section 20.5.10: SPI register map on page 611
0x4001 3000 - 0x4001 33FF	SPI1		
0x4001 2C00 - 0x4001 2FFF	SDIO		Section 21.9.16: SDIO register map on page 667
0x4001 2000 - 0x4001 23FF	ADC1		Section 11.12.16: ADC register map on page 240
0x4001 1400 - 0x4001 17FF	USART6		Section 19.6.8: USART register map on page 558
0x4001 1000 - 0x4001 13FF	USART1		
0x4001 0000 - 0x4001 03FF	TIM1		Section 12.4.21: TIM1 register map on page 314
0x4000 7000 - 0x4000 73FF	PWR	APB1	Section 5.5: PWR register map on page 90
0x4000 5C00 - 0x4000 5FFF	I2C3		Section 18.6.11: I2C register map on page 505
0x4000 5800 - 0x4000 5BFF	I2C2		
0x4000 5400 - 0x4000 57FF	I2C1		
0x4000 4400 - 0x4000 47FF	USART2		Section 19.6.8: USART register map on page 558
0x4000 4000 - 0x4000 43FF	I2S3ext		Section 20.5.10: SPI register map on page 611
0x4000 3C00 - 0x4000 3FFF	SPI3 / I2S3		
0x4000 3800 - 0x4000 3BFF	SPI2 / I2S2		
0x4000 3400 - 0x4000 37FF	I2S2ext		
0x4000 3000 - 0x4000 33FF	IWDG		Section 15.4.5: IWDG register map on page 426
0x4000 2C00 - 0x4000 2FFF	WWDG		Section 16.6.4: WWDG register map on page 433
0x4000 2800 - 0x4000 2BFF	RTC & BKP Registers		Section 17.6.21: RTC register map on page 471
0x4000 0C00 - 0x4000 0FFF	TIM5		Section 13.4.21: TIMx register map on page 374
0x4000 0800 - 0x4000 0BFF	TIM4		
0x4000 0400 - 0x4000 07FF	TIM3		
0x4000 0000 - 0x4000 03FF	TIM2		

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;
}

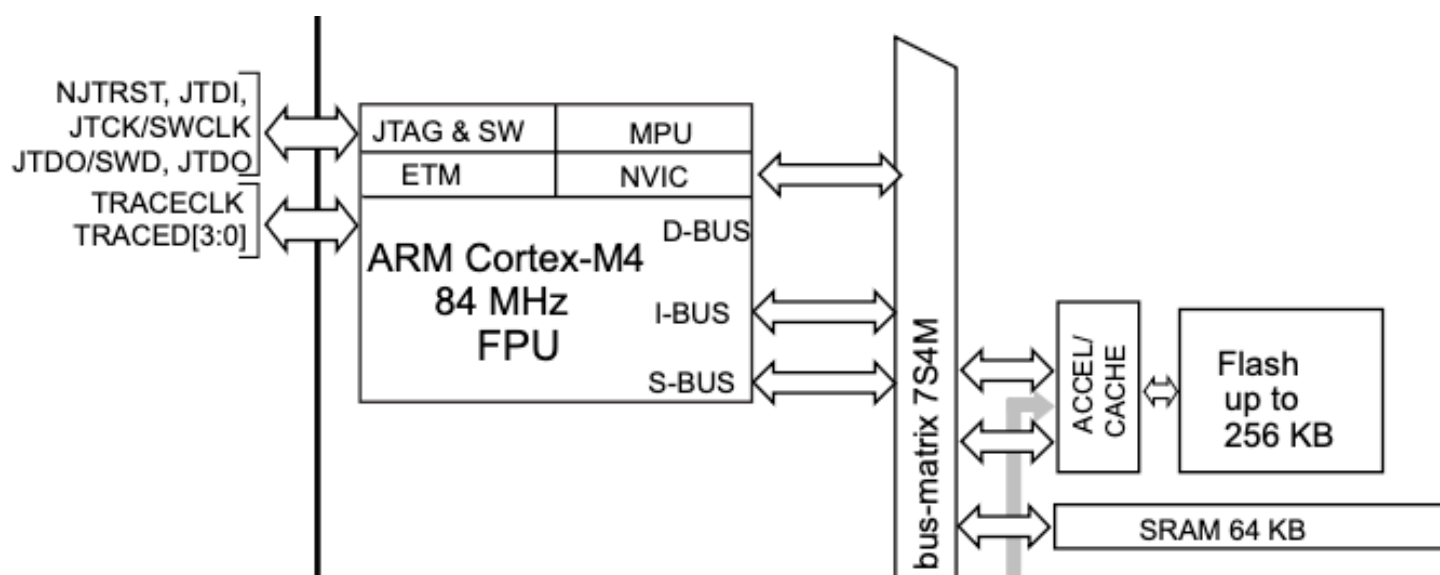
```

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.

In our next chapter we will discuss the vector table.

CHAPTER 2: VECTOR TABLE

The vector table is a table of system exception handler and interrupt address pointers.

VECTOR TABLE

Table 38. Vector table for STM32F401xB/CSTM32F401xD/E

Position	Priority	Type of priority	Acronym	Description	Address
	-	-	-	Reserved	0x0000 0000
	-3	fixed	Reset	Reset	0x0000 0004

Table 38. Vector table for STM32F401xB/CSTM32F401xD/E (continued)

Position	Priority	Type of priority	Acronym	Description	Address
	-2	fixed	NMI	Non maskable interrupt, Clock Security System	0x0000 0008
	-1	fixed	HardFault	All class of fault	0x0000 000C
	0	settable	MemManage	Memory management	0x0000 0010
	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000 0014
	2	settable	UsageFault	Undefined instruction or illegal state	0x0000 0018
	-	-	-	Reserved	0x0000 001C - 0x0000 002B
	3	settable	SVCall	System Service call via SWI instruction	0x0000 002C
	4	settable	Debug Monitor	Debug Monitor	0x0000 0030
		-	-	Reserved	0x0000 0034
	5	settable	PendSV	Pendable request for system service	0x0000 0038
	6	settable	Systick	System tick timer	0x0000 003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040
1	8	settable	EXTI16 / PVD	EXTI Line 16 interrupt / PVD through EXTI line detection interrupt	0x0000 0044
2	9	settable	EXTI21 / TAMP_STAMP	EXTI Line 21 interrupt / Tamper and TimeStamp interrupts through the EXTI line	0x0000 0048
3	10	settable	EXTI22 / RTC_WKUP	EXTI Line 22 interrupt / RTC Wakeup interrupt through the EXTI line	0x0000 004C
4	11	settable	FLASH	Flash global interrupt	0x0000 0050
5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000 0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000 0068
11	18	settable	DMA1_Stream0	DMA1 Stream0 global interrupt	0x0000 006C
12	19	settable	DMA1_Stream1	DMA1 Stream1 global interrupt	0x0000 0070
13	20	settable	DMA1_Stream2	DMA1 Stream2 global interrupt	0x0000 0074
14	21	settable	DMA1_Stream3	DMA1 Stream3 global interrupt	0x0000 0078

Table 38. Vector table for STM32F401xB/CSTM32F401xD/E (continued)

Position	Priority	Type of priority	Acronym	Description	Address
15	22	settable	DMA1_Stream4	DMA1 Stream4 global interrupt	0x0000 007C
16	23	settable	DMA1_Stream5	DMA1 Stream5 global interrupt	0x0000 0080
17	24	settable	DMA1_Stream6	DMA1 Stream6 global interrupt	0x0000 0084
18	25	settable	ADC	ADC1 global interrupts	0x0000 0088
23	30	settable	EXTI9_5	EXTI Line[9:5] interrupts	0x0000 009C
24	31	settable	TIM1_BRK_TIM9	TIM1 Break interrupt and TIM9 global interrupt	0x0000 00A0
25	32	settable	TIM1_UP_TIM10	TIM1 Update interrupt and TIM10 global interrupt	0x0000 00A4
26	33	settable	TIM1_TRG_COM_TIM11	TIM1 Trigger and Commutation interrupts and TIM11 global interrupt	0x0000 00A8
27	34	settable	TIM1_CC	TIM1 Capture Compare interrupt	0x0000 00AC
28	35	settable	TIM2	TIM2 global interrupt	0x0000 00B0
29	36	settable	TIM3	TIM3 global interrupt	0x0000 00B4
30	37	settable	TIM4	TIM4 global interrupt	0x0000 00B8
31	38	settable	I2C1_EV	I ² C1 event interrupt	0x0000 00BC
32	39	settable	I2C1_ER	I ² C1 error interrupt	0x0000 00C0
33	40	settable	I2C2_EV	I ² C2 event interrupt	0x0000 00C4
34	41	settable	I2C2_ER	I ² C2 error interrupt	0x0000 00C8
35	42	settable	SPI1	SPI1 global interrupt	0x0000 00CC
36	43	settable	SPI2	SPI2 global interrupt	0x0000 00D0
37	44	settable	USART1	USART1 global interrupt	0x0000 00D4
38	45	settable	USART2	USART2 global interrupt	0x0000 00D8
40	47	settable	EXTI15_10	EXTI Line[15:10] interrupts	0x0000 00E0
41	48	settable	EXTI17 / RTC_Alarm	EXTI Line 17 interrupt / RTC Alarms (A and B) through EXTI line interrupt	0x0000 00E4
42	49	settable	EXTI18 / OTG_FS_WKUP	EXTI Line 18 interrupt / USB On-The-Go FS Wakeup through EXTI line interrupt	0x0000 00E8
47	54	settable	DMA1_Stream7	DMA1 Stream7 global interrupt	0x0000 00FC
49	56	settable	SDIO	SDIO global interrupt	0x0000 0104
50	57	settable	TIM5	TIM5 global interrupt	0x0000 0108
51	58	settable	SPI3	SPI3 global interrupt	0x0000 010C

Table 38. Vector table for STM32F401xB/CSTM32F401xD/E (continued)

Position	Priority	Type of priority	Acronym	Description	Address
56	63	settable	DMA2_Stream0	DMA2 Stream0 global interrupt	0x0000 0120
57	64	settable	DMA2_Stream1	DMA2 Stream1 global interrupt	0x0000 0124
58	65	settable	DMA2_Stream2	DMA2 Stream2 global interrupt	0x0000 0128
59	66	settable	DMA2_Stream3	DMA2 Stream3 global interrupt	0x0000 012C
60	67	settable	DMA2_Stream4	DMA2 Stream4 global interrupt	0x0000 0130
67	74	settable	OTG_FS	USB On The Go FS global interrupt	0x0000 014C
68	75	settable	DMA2_Stream5	DMA2 Stream5 global interrupt	0x0000 0150
69	76	settable	DMA2_Stream6	DMA2 Stream6 global interrupt	0x0000 0154
70	77	settable	DMA2_Stream7	DMA2 Stream7 global interrupt	0x0000 0158
71	78	settable	USART6	USART6 global interrupt	0x0000 015C
72	79	settable	I2C3_EV	I ² C3 event interrupt	0x0000 0160
73	80	settable	I2C3_ER	I ² C3 error interrupt	0x0000 0164
81	88	Settable	FPU	FPU global interrupt	0x0000 0184
84	91	settable	SPI4	SPI 4 global interrupt	0x0000 0190

Let's turn to the vector table within our reference manual. In 10.2 we find our vector table on pages 203-205.

The first column is position or IRQ number. The system exceptions have as fixed position so they will be empty.

The interrupts have positions that start at 0 starting with the Window Watchdog Interrupt.

The second column is priority. The lower the numerical value means the higher priority.

The third column is the type of priority whether it can be changed by software or not.

The other columns are self explanatory.

The literal flash we can think of like a little on-board hard drive. It, on disk, starts at 0x00000000 and gets mapped on boot to 0x08000000.

The vector table then starts with 4 bytes for the master stack pointer and then at 0x00000004 which maps to 0x08000004 we see the reset handler which will ultimately call our main function in C.

There are other system exceptions that by default all map to the reset handler however you can change them in software.

Regarding the interrupts, you can also implement these in software which we will do later in this course.

FLASH MEMORY PLACEMENT

Flash starts at 0x08000000 as we discussed in the last chapter and specifically 0x08000000 is where the stack start address exists on boot.

+-----+	
	FLASH
+-----+	
	0x08000193 End of Vector Tbl
+-----+	
	... (IRQ/exception)
+-----+	
	0x08000190 SPI4 IRQ Handler
+-----+	
	0x08000184 FPU IRQ Handler
+-----+	
	... (IRQ/exception)
+-----+	
	0x0800000C HardFault Handler
+-----+	
	0x08000008 NMI Handler
+-----+	
	0x08000004 Reset Handler
+-----+	
	0x08000000 Initial SP Value
+-----+	

Total size: 404 bytes (0x194)

- Each entry: 4 bytes
- Number of entries: 101

We see 4 bytes of the initial stack pointer starting at 0x08000000 and going all the way to 0x08000193 which covers 404 bytes of our vector table.

At 0x08000000, there is a function **g_pfnVectors** to which the vector table is stored.

CHAPTER 3: BOOT

When the MCU boots, it maps 0x00000000 to 0x08000000 and sets up the vector table.

G_PFNVECTORS

In our last chapter we covered the vector table as this is the first thing to get mapped at 0x08000000.

__DO_GLOBAL_DTORS_AUX

Lets examine what comes after the vector table in the running firmware starting at 0x08000194.

```
disas 0x08000194
Dump of assembler code for function __do_global_dtors_aux:
0x08000194 <+0>:      push    {r4, lr}
0x08000196 <+2>:      ldr     r4, [pc, #20]    ; (0x80001ac <__do_global_dtors_aux+24>)
0x08000198 <+4>:      ldrb    r3, [r4, #0]
0x0800019a <+6>:      cbnz    r3, 0x80001aa <__do_global_dtors_aux+22>
0x0800019c <+8>:      ldr     r3, [pc, #16]    ; (0x80001b0 <__do_global_dtors_aux+28>)
0x0800019e <+10>:     cbz     r3, 0x80001a6 <__do_global_dtors_aux+18>
0x080001a0 <+12>:     ldr     r0, [pc, #16]    ; (0x80001b4 <__do_global_dtors_aux+32>)
0x080001a2 <+14>:     nop.w
0x080001a6 <+18>:     movs    r3, #1
0x080001a8 <+20>:     strb    r3, [r4, #0]
0x080001aa <+22>:     pop     {r4, pc}
0x080001ac <+24>:     movs    r0, r0
0x080001ae <+26>:     movs    r0, #0
0x080001b0 <+28>:     movs    r0, r0
0x080001b2 <+30>:     movs    r0, r0
0x080001b4 <+32>:     lsls    r0, r0, #10
0x080001b6 <+34>:     lsrs    r0, r0, #32
End of assembler dump.
```

This is the `__do_global_dtors_aux` function, which is part of the C runtime initialization/cleanup code that gets automatically inserted by the GCC toolchain. Let me break down this ARM Thumb-2 assembly code in detail:

Function Purpose

`__do_global_dtors_aux` (do global destructors auxiliary) is responsible for calling C++ destructors for global/static objects when the program exits. However, on embedded systems like the STM32F401RE, this is often a stub since programs typically don't "exit" in the traditional sense.

Detailed Assembly Analysis

Function Prologue

0x08000194 <+0>: push {r4, lr}

Saves register r4 and the link register (lr) on the stack.

Standard function prologue to preserve caller's context/

Main Logic Block

0x08000196 <+2>: ldr r4, [pc, #20] ; (0x80001ac
<__do_global_dtors_aux+24>)

0x08000198 <+4>: ldrb r3, [r4, #0]

0x0800019a <+6>: cbnzr3, 0x80001aa <__do_global_dtors_aux+22>

ldr r4, [pc, #20]: Loads a pointer from address 0x80001ac into r4 (PC-relative addressing)

ldrb r3, [r4, #0]: Loads a byte from the address pointed to by r4 into r3

cbnz r3, 0x80001aa: "Compare and Branch if Non-Zero" - if the byte is non-zero, jump to the exit

This implements a "guard variable" pattern - checking if destructors have already been called.

Destructor Call Logic

0x0800019c <+8>: ldr r3, [pc, #16] ; (0x80001b0
<__do_global_dtors_aux+28>)

0x0800019e <+10>: cbz r3, 0x80001a6 <__do_global_dtors_aux+18>

0x080001a0 <+12>: ldr r0, [pc, #16] ; (0x80001b4
<__do_global_dtors_aux+32>)

0x080001a2 <+14>: nop.w

Loads another pointer from 0x80001b0.

If this pointer is zero, skip the destructor calls.

If non-zero, load an argument into r0 and prepare for a function call.

The **nop.w** suggests there should be a function call here, but it's been optimized out or is a placeholder

Set Guard and Exit

```
0x080001a6 <+18>:  movs    r3, #1
```

```
0x080001a8 <+20>:  strb r3, [r4, #0]
```

```
0x080001aa <+22>:  pop {r4, pc}
```

Sets the guard variable to 1 (marking destructors as “called”).

Restores r4 and returns (popping lr into pc).

Literal Pool

```
0x080001ac <+24>:  movs    r0, r0    ; Actually: .word guard_variable_address
```

```
0x080001ae <+26>:  movs    r0, #0    ; Actually: .word 0x00000000
```

```
0x080001b0 <+28>:  movs    r0, r0    ; Actually: .word destructor_list_ptr
```

```
0x080001b2 <+30>:  movs    r0, r0    ; Actually: .word 0x00000000
```

```
0x080001b4 <+32>:  lsls    r0, r0, #10 ; Actually: .word some_parameter
```

```
0x080001b6 <+34>:  lsrs    r0, r0, #32 ; Actually: .word 0x00000000
```

The disassembler is incorrectly interpreting these as instructions - they're actually 32-bit literal values stored in the literal pool that the PC-relative loads reference.

Why This Appears After the Vector Table

Startup Sequence: After reset, the processor jumps to the reset handler in the vector table.

Runtime Initialization: The reset handler calls various initialization functions, including this one.

Memory Layout: The linker places this code immediately after the vector table in flash memory.

Embedded Context: On bare-metal systems, this function often does nothing (as seen here with the optimized-out destructor calls) since there's typically no program termination.

This is essentially boilerplate code that ensures proper C++ semantics are maintained, even in embedded contexts where global destructors may never actually be called.

FRAME_DUMMY

Lets examine what comes next in the running firmware starting at 0x0800001B8.

```
disas 0x080001b8
Dump of assembler code for function frame_dummy:
0x080001b8 <+0>:    push    {r3, lr}
0x080001ba <+2>:    ldr     r3, [pc, #12]    ; (0x80001c8 <frame_dummy+16>)
0x080001bc <+4>:    cbz     r3, 0x80001c6 <frame_dummy+14>
0x080001be <+6>:    ldr     r1, [pc, #12]    ; (0x80001cc <frame_dummy+20>)
0x080001c0 <+8>:    ldr     r0, [pc, #12]    ; (0x80001d0 <frame_dummy+24>)
0x080001c2 <+10>:   nop.w
0x080001c6 <+14>:   pop     {r3, pc}
0x080001c8 <+16>:   movs    r0, r0
0x080001ca <+18>:   movs    r0, r0
0x080001cc <+20>:   movs    r4, r0
0x080001ce <+22>:   movs    r0, #0
0x080001d0 <+24>:   lsls    r0, r0, #10
0x080001d2 <+26>:   lsrs    r0, r0, #32
End of assembler dump.
```

This is the `frame_dummy` function, another piece of C runtime initialization code that's part of the GCC toolchain's startup sequence. Let me break down this ARM Thumb-2 assembly code:

Function Purpose

`frame_dummy` is responsible for registering frame unwinding information for exception handling and debugging. It's the counterpart to `__do_global_dtors_aux` but deals with constructor registration rather than destructor cleanup.

Detailed Assembly Analysis

Function Prologue

```
0x080001b8 <+0>:    push    {r3, lr}
```

Saves r3 (for stack alignment) and the link register (lr).

Standard ARM calling convention prologue.

Main Logic Block

0x080001ba <+2>: ldr r3, [pc, #12] ; (0x80001c8 <frame_dummy+16>)

0x080001bc <+4>: cbz r3, 0x80001c6 <frame_dummy+14>

ldr r3, [pc, #12]: Loads a pointer from the literal pool at offset +16 (0x80001c8)

cbz r3, 0x80001c6: "Compare and Branch if Zero" - if the loaded value is zero, skip to the exit

This checks if there's a frame registration function available.

Frame Registration Setup

0x080001be <+6>: ldr r1, [pc, #12] ; (0x80001cc <frame_dummy+20>)

0x080001c0 <+8>: ldr r0, [pc, #12] ; (0x80001d0 <frame_dummy+24>)

0x080001c2 <+10>: nop.w

Loads two more values from the literal pool into r1 and r0

These would be arguments for a frame registration function call

The **nop.w** is a 32-bit NOP - this is where a function call (bl instruction) would normally be, but it's been optimized out

Function Exit

0x080001c6 <+14>: pop {r3, pc}

Restores r3 and returns by popping lr into pc.

Standard function epilogue.

Literal Pool (Misinterpreted by Disassembler)

0x080001c8 <+16>: movs r0, r0 ; Actually: .word frame_register_func_ptr

0x080001ca <+18>: movs r0, r0 ; Actually: upper 16 bits of above

0x080001cc <+20>: movs r4, r0 ; Actually: .word frame_info_ptr

0x080001ce <+22>: movs r0, #0 ; Actually: upper 16 bits of above

0x080001d0 <+24>: lsls r0, r0, #10 ; Actually: .word frame_object_ptr

0x080001d2 <+26>: lsrs r0, r0, #32 ; Actually: upper 16 bits of above

Again, the disassembler is incorrectly showing these as instructions when they're actually 32-bit literal values in the literal pool.

What This Function Normally Does

In a full C++ environment, **frame_dummy** would:

Check if frame registration is available - via the function pointer in the literal pool.

Register exception handling frames - for C++ exception unwinding.

Set up debugging information - for stack traces and debugging.

Why It's Optimized Out Here

On the STM32F401RE bare-metal system:

No Exception Handling: C++ exceptions are typically disabled in embedded systems.

No Dynamic Registration: Frame information is usually statically linked.

Size Optimization: The linker removes unused functionality.

Real-time Constraints: Exception handling overhead is undesirable.

Startup Sequence Context

This function is called during the C runtime initialization sequence:

Reset Handler (from vector table).

Hardware Initialization (clocks, memory, etc.).

C Runtime Setup:

frame_dummy ← We are here.

Global constructor calls.

main function call.

The function exists to maintain compatibility with the standard C++ runtime model, even though its actual functionality is stripped out for embedded use. It's essentially a stub that ensures the startup sequence completes properly without breaking the expected call chain.

MAIN

Finally at 0x080001D4 we have **main**.

In our next chapter we will cover the MCU clock.

CHAPTER 4: CLOCK

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

The MCU 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. Let's start with GPIOA which is a general-purpose input-output series of registers.

We first need to turn to page 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.


```
1  #include <stm32f401xe.h>
2
3  int main(void) {
4      // Initialize the AHB1 system clock for GPIOA
5      RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // RM p.118
6
7      // Main loop
8      while (1) {
9      }
10
11     return 0;
12 }
```

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

CHAPTER 4: REVERSING CLOCK

In this chapter we will reverse engineer the clock program. Lets review the code.

Lets review the code we have currently.

```
1  #include <stm32f401xe.h>
2
3  int main(void) {
4      // Initialize the AHB1 system clock for GPIOA
5      RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // RM p.118
6
7      // Main loop
8      while (1) {
9      }
10
11     return 0;
12 }
```

CODE WALKTHROUGH

We will start with line 5. We see `RCC->AHB1ENR` where `RCC` is a pointer to the Reset and Clock Control peripheral structure defined in the include on line 1.

The `AHB1ENR` stands for the AHB1 peripheral clock enable register. This register controls the clock gating for peripherals on the AHB1 bus.

We see the `|=` which means we are going to set the `RCC_AHB1ENR_GPIOAEN` bit and only that bit and assign back into the pointer to `RCC->AHB1ENR`.

Let's command click on `RCC` and it will take us to the definition within `stm32f401xe.h`.

```
786 #define RCC ((RCC_TypeDef *) RCC_BASE)
```

We see `RCC` is part of a larger struct and we can further click on `RCC_BASE`.

```
698 #define RCC_BASE (AHB1PERIPH_BASE + 0x3800UL)
```

Here we see `RCC_BASE` is the `AHB1PERIPH_BASE + 0x3800` and if we review our reference manual page 38, table 1, we see the AHB1 base starts at 0x40020000 and RCC is at the 0x3800 offset.

Let's also command click on `AHB1PERIPH_BASE`.

```
#define AHB1PERIPH_BASE    (PERIPH_BASE + 0x0020000UL)
```

We can see on page 38 of the reference manual that AHB1 is 0x20000 offset from the base.

Let's finally command click on `PERIPH_BASE`.

```
637 #define PERIPH_BASE    0x40000000UL
```

We can see on page 39 of the reference manual that the base of the peripherals is at 0x40000000 starting with the TIM2 peripheral on the APB1 bus.

Let's also command click on the `RCC_AHB1ENR_GPIOAEN`.

```
4265 #define RCC_AHB1ENR_GPIOAEN_Pos    (0U)
4266 #define RCC_AHB1ENR_GPIOAEN_Msk    (0x1UL << RCC_AHB1ENR_GPIOAEN_Pos) /*!< 0x00000001 */
4267 #define RCC_AHB1ENR_GPIOAEN    RCC_AHB1ENR_GPIOAEN_Msk
```

Here we see that we start with `RCC_AHB1ENR_GPIOAEN_Pos` of 0 and `<< 0x1` therefore `RCC_AHB1ENR_GPIOAEN` will be 0x1.

CODE REVERSING

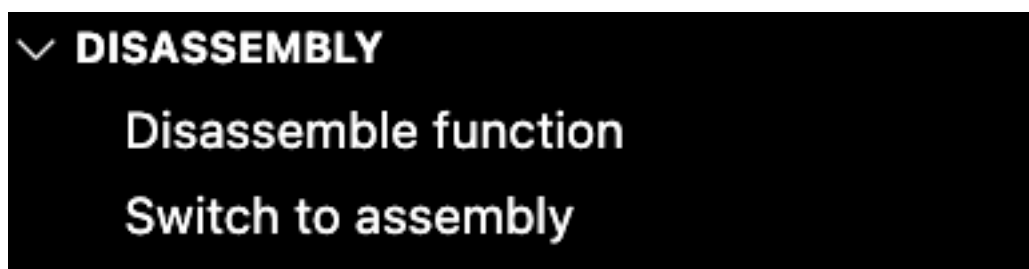
We will put a breakpoint on line 5 and debug within VSCode and run.

```
● 5    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // RM p.118
```

We see that we successfully broke on line 5.

```
🔍 5    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // RM p.118
```

In the disassembly window on the left, let's click on Switch to assembly.

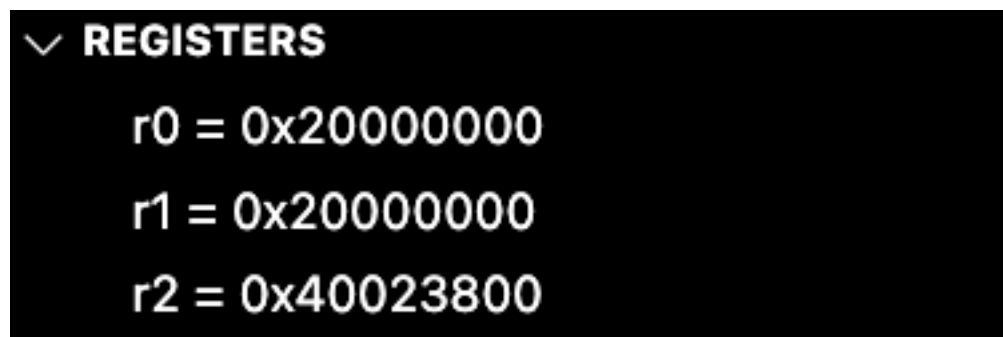


Let's examine our Assembler code...

```
1 0x080001d4: 02 4a      ldr r2, [pc, #8] ; (0x80001e0 <main+12>)
2 0x080001d6: 13 6b      ldr r3, [r2, #48] ; 0x30
3 0x080001d8: 43 f0 01 03  orr.w  r3, r3, #1
4 0x080001dc: 13 63      str r3, [r2, #48] ; 0x30
5 0x080001de: fe e7      b.n 0x80001de <main+10>
6 0x080001e0: 00 38      subs  r0, #0
7 0x080001e2: 02 40      ands  r2, r0
8
```

If you do not understand ARM Assembler, I have written a free Embedded Assembler book to get you a primer as it is located on my GitHub <https://github.com/mytechnotalent/Embedded-Assembler>.

We start with the `ldr r2, [pc, #8]` instruction. The line is highlighted as this means this has not yet been executed. This will take what is pointed to at the program counter with an offset of 8 and then store that value into R2. Let's step into and see what R2 will then hold after the step has been taken.

A screenshot of a software interface with a dark background. At the top, there is a dropdown menu labeled 'REGISTERS' with a downward arrow. Below it, the values of three registers are displayed: r0 = 0x20000000, r1 = 0x20000000, and r2 = 0x40023800.

Here we see R2 holds 0x40023800 and we know this is the RCC base register address as we can see this on page 38 of the reference manual.

```

1  0x080001d4: 02 4a      ldr r2, [pc, #8]      ; (0x80001e0 <main+12>)
2  0x080001d6: 13 6b      ldr r3, [r2, #48]     ; 0x30
3  0x080001d8: 43 f0 01 03    orr.w  r3, r3, #1
4  0x080001dc: 13 63      str r3, [r2, #48]     ; 0x30
5  0x080001de: fe e7      b.n 0x80001de <main+10>
6  0x080001e0: 00 38      subs  r0, #0
7  0x080001e2: 02 40      ands  r2, r0

```

We are now about to execute `ldr r3, [r2, #48]` which means load the pointer within R2 with an offset of 48 or 0x30 and store it into R3. An LDR instruction takes what is right of the command and stores it inside what is left of the comma.

Before we take this step, let's see what is at that address in the debug console.

```

→ x/x 0x40023830
0x40023830: 0x00000000

```

We see that there is currently nothing there so when we take this step R3 should be 0x00000000. Let's take the step and review R3.

```

✓ REGISTERS

r0 = 0x20000000
r1 = 0x20000000
r2 = 0x40023800
r3 = 0x00000000

```

We have confirmed that R3 now holds 0x00000000.

```

1  0x080001d4: 02 4a      ldr r2, [pc, #8]      ; (0x80001e0 <main+12>)
2  0x080001d6: 13 6b      ldr r3, [r2, #48]     ; 0x30
3  0x080001d8: 43 f0 01 03    orr.w  r3, r3, #1
4  0x080001dc: 13 63      str r3, [r2, #48]     ; 0x30
5  0x080001de: fe e7      b.n 0x80001de <main+10>
6  0x080001e0: 00 38      subs  r0, #0
7  0x080001e2: 02 40      ands  r2, r0

```


We now are about to execute `orr.w r3, r3, #1` which means we will OR 1 with what is currently in R3 and then store the result in R3. We know R3 has 0x00000000 in it so lets manually do the OR math.

```
0000 0000 0000 0000 0000 0000 0000 0000 R3
+ 0000 0000 0000 0000 0000 0000 0000 0000 #1
```

```
0000 0000 0000 0000 0000 0000 0000 0001
```

Now we will step and see R3 will in fact be 0x00000001. Remember the above OR math shows binary and every digit is a bit. Every 4 bits is a nibble. Every 8 bits is a byte. Every 2 nibbles are a byte.

```
✓ REGISTERS

r0 = 0x20000000
r1 = 0x20000000
r2 = 0x40023800
r3 = 0x00000001
```

We can now continue and review our Assembler.

```
1  0x080001d4: 02 4a      ldr r2, [pc, #8]      ; (0x80001e0 <main+12>)
2  0x080001d6: 13 6b      ldr r3, [r2, #48]     ; 0x30
3  0x080001d8: 43 f0 01 03  orr.w  r3, r3, #1
4  0x080001dc: 13 63      str r3, [r2, #48]     ; 0x30
5  0x080001de: fe e7      b.n 0x80001de <main+10>
6  0x080001e0: 00 38      subs  r0, #0
7  0x080001e2: 02 40      ands  r2, r0
```

We are about to execute the STR instruction and unlike the LDR, it works from left to right. Here we are going to store what is in R3 which is 0x00000001 into the pointer of R2 with an offset of 48 or 0x30.

When I say pointer, you can see it referenced by the [] brackets. That means work on what is pointed to within that context.

Here we see R2 holds 0x40023800. We need to add an 0x30 offset. Let's use the debug console and see what 0x40023830 currently has inside it.

```
→ x/x 0x40023830
0x40023830: 0x00000000
```

We can see that it currently has 0x00000000 meaning GPIOA is not enabled. Let's step again and re-examine.

```
1 0x080001d4: 02 4a      ldr r2, [pc, #8]      ; (0x80001e0 <main+12>)
2 0x080001d6: 13 6b      ldr r3, [r2, #48]     ; 0x30
3 0x080001d8: 43 f0 01 03    orr.w r3, r3, #1
4 0x080001dc: 13 63      str r3, [r2, #48]     ; 0x30
5 0x080001de: fe e7      b.n 0x80001de <main+10>
6 0x080001e0: 00 38      subs r0, #0
7 0x080001e2: 02 40      ands r2, r0
```

```
→ x/x 0x40023830
0x40023830: 0x00000001
```

Here we see that we successfully set 0x1 into the AHB1ENR register therefore enabling GPIOA.

The next line highlighted is simply the infinite loop. In our next chapter we will discuss GPIO's in more detail.