6 A Tourist's Phrasebook for Reversing Embedded ARM in the Dialect of the Cortex M Series

by Travis Goodspeed and Ryan Speers

Ahoy there, neighbor!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the architecture of smaller devices as quickly as possible, with a minimum of fuss and formality.

Those of you who have already worked with ARM might find it to be a useful refresher, while those of you new to the architecture will find that it isn't really as strange as you've been led to believe. If you've already reverse engineered binaries for any platform, even x86 Windows applications, you'll soon feel right at home.

We've written this guide with STM32 devices for specific examples, but with minor differences it applies well enough to the Cortex M series as a whole. These devices generally have a megabyte or less of Flash and at most a few hundred kilobytes of RAM. By and large, they only run the Thumb2 instruction set, without support for the older AARCH32 instruction set. For larger ARM chips, such as those used in smartphones and tablets, you might be better served by a different introduction.

6.1 At a Glance

Common Models
STM32, EFM32

Architecture
32-bit registers
16-bit and 32-bit Thumb(2) instructions

Registers
R15: Program Counter
R14: Link Register
R13: Stack Pointer
R0 to R12: General Use

6.2 Basics of the Instruction Set

Back in the day, ARM used fixed-width 32-bit RISC instructions. Like the creation of the world, this was widely regarded as a mistake, and many angry people wrote comments complaining that it was a waste of space, and that RISC wouldn't "change everything." These instructions were always 32-bit word aligned, so the lowest two bits of the Program Counter (R15) were always zero.

Larger ARM chips, such as those in an early smartphone, support two instructions sets. If the least significant bit of the program counter is clear (0), then the 32-bit instruction set is used, whereas if that bit is set (1), the chip will use a 16-bit instruction set called Thumb. Registers are still 32 bits wide, but the instructions themselves are only a half-word. They must be half-word aligned.

Because Thumb instructions have fewer bits to spare, code in larger ARM machines will switch between ARM and Thumb as it is convenient. You can see this in the least significant bit of a function pointer, where an ARM function's address will be even, while a Thumb function's address will be odd.

The Cortex M3 devices speak a slimmer dialect than the big-iron ARM chips. This dialect drops the 32-bit wide instruction set entirely, supporting only Thumb and Thumb2 instructions.[9] Because of this, all functions and all interrupt handlers are referred to by *odd* addresses, which are actually the address of the byte *after* the real starting address! If you see a call to `0x08005615`, that is really a call to the Thumb code at `0x08005614`.

6.3 Registers and Calling Convention

Arguments are passed to the child function from R0 to R3. R4 to R11 hold local variables, and the child function *must* restore them before returning to the parent function. Values are returned in R0 to R3, and these registers are not preserved by the child.

Much like in PowerPC and very unlike x86, the Link Register (R14, a.k.a. LR) holds the return address. A leaf function, having no children, might never write its return pointer to the stack. The `BL` instruction automatically moves the old Program Counter into the Link Register when calling a child, so parent functions must manually save R14 before calling children. The return instruction, `BLR`, functions by moving R14 (LR) into R15 (PC).

[9] Thumb2 instructions run from Thumb mode. The only thing new about them is that they can be longer than 16 bits, so your disassembler might be slightly confused about their starting position.

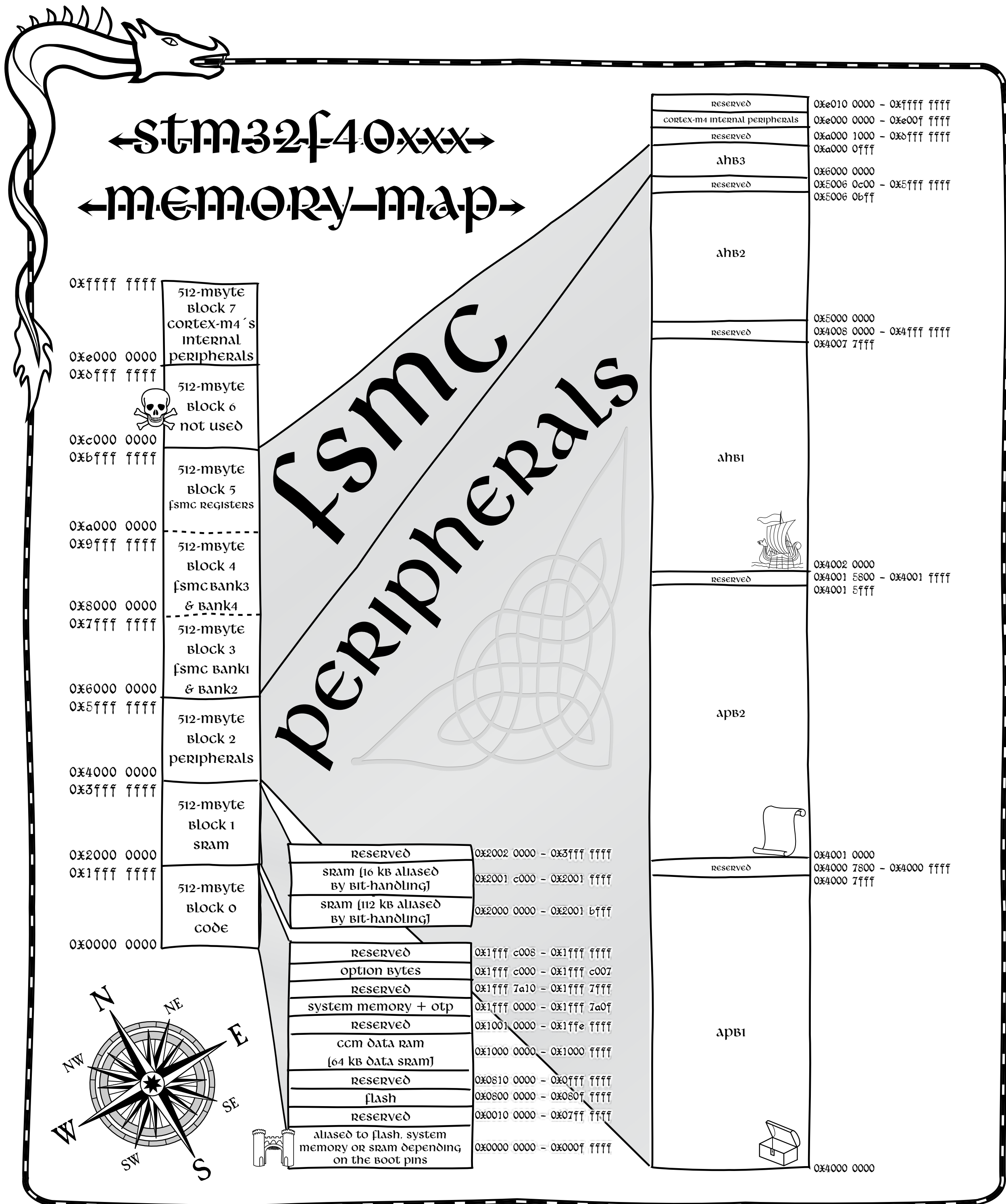

Figure 3 – STM32F40xxx Memory Map

6.4 Memory Map

Figure 3 shows the memory layout of the STM32F405, a Cortex M4 device. Study this map for a moment, before we go on to how to use it in your adventure!

Because Cortex M devices have four gigabytes of address space but hardly a megabyte of Flash, they keep functionally different parts of memory at very different addresses.

Code memory is officially the range from `0x00000000` to `0x1FFFFFFF`, but in nearly all cases, you'll find that Flash is mapped to begin at `0x0800-0000`. When reverse engineering an application, you'll find that it's either written here or a few dozens of kilobytes later, to leave room for a bootloader.

SRAM is usually mapped to begin at `0x2000-0000`, so it's safe to assume that any read or write to an absolute address in this region is a global variable, and also that the stack and heap fit somewhere in this range. Unlike a desktop application, which loads its initial globals directly into a `.data` segment, an embedded application must manually initialize its data variables, possibly by copying a large chunk from Flash into SRAM.

Peripheral memory begins at `0x40000000`. Both because peripherals are most often referred to by an explicit address, and because Flash comes with no linking systems or system calls, reads and writes to this region are a gold mine for a reverse engineer!

System control registers are at `0xE0000000`. These are used to do things like moving the interrupt table or reading the chip's model number.

6.5 Making Sense of Pointers

Let us teach you some nifty tricks about pointers in Thumb machines.

Back when ARM was first designed, 32-bit fixed-width instructions with 32-bit alignment were all the rage, and all the cool kids (POWER, SPARC, Alpha) used them. Later on, when the Thumb instruction set was being designed, its designers chose 16-bit instructions that could be mapped back to the same 32-bit core. The CPU would fetch a 32-bit ARM instruction if the least-significant bit of the program counter were even, and a 16-bit Thumb instruction if the program counter were odd.

But these Cortex chips generally ship just Thumb and Thumb2, without backward compatibility to 32-bit ARM instructions. So the trick, which you can try in the next section, is that data pointers are always even and instruction (function) pointers are always odd.

6.6 Making Sense of the Interrupt Table

Let's take a look at the interrupt table from the beginning of a Cortex M firmware image. These are 32-bit little endian addresses, which are to be read backwards.

```
0000000 30 14 00 20    21 41 00 08
        39 57 00 08    3d 57 00 08
0000010 41 57 00 08    45 57 00 08
        49 57 00 08    00 00 00 00
0000020 00 00 00 00    00 00 00 00
        00 00 00 00    51 57 00 08
0000030 4d 57 00 08    00 00 00 00
        55 57 00 08    59 57 00 08
0000040 ...
```

Note that the first word, `0x20001430`, is in the SRAM region; this is because the first word of a Cortex M interrupt table is the initialization value for the Stack Pointer (R13). The second word, `0x0800-4121`, is the initialization value for the Program Counter (R15), so we know the entry point of the application is Thumb2 code starting at `0x08004120`.

Except for some reserved (zeroed) words, the handler addresses are all in Flash memory and represent the interrupt handler functions. We can look up the meaning of each handler in the specific chip's programming guide, then chase the ones that are most relevant. For example, if we are reverse engineering a USB device, powered by an STM32F3xx, the STM32F37xx reference manual tells us that the interrupts at offsets `0x000000D8` and `0x0000001C` handle USB events. These might be good handlers to reverse early in the process.

6.7 Loading into IDA Pro or Radare2

To load the application into IDA Pro or Radare2, you generally need to know the loading point and the locations of some other memories.

The loading point will be at or near `0x08000000`, depending upon whether a bootloader comes before your image. If you are working from a JTAG dump, just use the address the image came from. If you are working from a `.dfu` (Device Firmware Update) file, it will contain a loading address in its header metadata.

When given a raw dump without a starting address, disassemble the instructions and try to find a loading address at which the interrupt handlers line up. (The interrupt vector table is usually at 0x08000000 at boot, but it can be moved to a new address by software.)

6.8 Making Sense of the Peripherals

The Cortex M3 contains two peripheral regions. At 0x40000000, you will find the most useful ones for reverse engineering applications, such as UART and USB controllers, General Purpose IO (GPIO), and other devices. Unfortunately, these peripherals are not generic to the Cortex M3 as an architecture; rather, they are specific to each individual chip.

Supposing you are reverse engineering an application for the STM32F3xx series, you would download the Peripheral Support Library for that chip from its manufacturer and eventually find yourself reading `stm32f30x.h`. For other chips, there are similar headers, each of which is written around C structs for register groups and preprocessor definitions for peripheral base addresses and offsets.

Suppose we know from reverse engineering a circuit board that USART2 is used by our target application to send packets to a radio chip, and we would like to search for all functions that use this peripheral. Working backwards, we find the following relevant lines in `stm32f30x.h`.

```
//Abbreviated USART register struct.
typedef struct{
  __IO uint32_t CR1;      //+0x00
  __IO uint32_t CR2;
  __IO uint32_t CR3;
  __IO uint16_t BRR;
  uint16_t  RESERVED1;
  __IO uint16_t GTPR;
  uint16_t  RESERVED2;
  __IO uint32_t RTOR;
  __IO uint16_t RQR;
  uint16_t  RESERVED3;
  __IO uint32_t ISR;
  __IO uint32_t ICR;
  __IO uint16_t RDR;      //+0x24 RX Data Reg
  uint16_t  RESERVED4;
  __IO uint16_t TDR;      //+0x28 TX Data Reg
  uint16_t  RESERVED5;
} USART_TypeDef;

//USART location definitions.
#define USART2                                  \
        ((USART_TypeDef *) USART2_BASE)
#define USART2_BASE                             \
        (APB1PERIPH_BASE + 0x00004400)
#define APB1PERIPH_BASE                         \
        PERIPH_BASE
#define PERIPH_BASE                             \
        ((uint32_t)0x40000000)
```

This means that USART2's data structure is located at 0x40004400. From the `USART_TypeDef` structure, we know that data is received from USART2 by reading 0x40004424 and written to USART2 by writing to 0x40004428! Searching for these addresses ought to easily find us the read and write functions for that port.

6.9 Other Oddities

Please note that this guide has left out some features unique to the STM32 series, and that each chip has its own little quirks. You'll find different memory maps on each implementation, and anything that looks confusing is likely worth spending more time to understand.

For example, some ARM devices offer Core-Coupled Memory (CCM), which is SRAM that's wired directly to the CPU's internal data bus rather than to the main memory bus of the chip. This makes fetches lightning fast, but has the complications that the memory is unusable for DMA or code fetches. Care for a non-executable stack, anyone?

Another quirk is that many devices map the same physical memory to multiple virtual locations. In some high-performance code, the use of both cached and uncached memory can allow for more efficient operation.

Additionally, address zero often contains a duplicate of the boot memory, which is usually Flash but might be executable SRAM. Presumably this was done to allow for code that has compatible immediate addresses when booting from either memory, but PoC||GTFO 10:8 describes a nifty little jailbreak that relies on dumping the 48K recovery bootloader of an STM32F405 chip out of Flash through a null-pointer read.

– – – –   – – –   – – – –

We hope that you've enjoyed this friendly little guide to the Cortex M3, and that you'll keep it handy when reverse engineering firmware from that platform.