Introduction to Microcontrollers Notes

James Gowans

August 31, 2014

Licence

Contents

1 System Overview

1.1 What is a Microcontroller?

The microcontroller can be understood by comparing it to something you are already very familiar with: the computer. Both a microcontroller and a computer can be modeled as a black box which takes in data and instructions, performs processing, and provides output. In order to do this, a micro has some of the same internals as a computer, shown graphically in Figure 1.1 and discussed now:

- CPU: The section of the microcontroller which does the processing. It executes instructions which allows it to do arithmetic and logic operations, amongst other forms of operations.
- Volatile memory (RAM:) This is general purpose memory. It can be used for storing whatever you want to store in it. Typically it stores variables which are created or changed during the course of execution of a program.
- Non-volatile memory (Flash): This non-volatile memory is used to store any date which must not be lost when the power to the micro is removed. Typically this would include the program code and any constants or initial values of data.
- Ports: Interfaces for data to move in and out of the micro. This allow it to communicate with the outside world.

These resources are typically orders of magnitude smaller or a micro than on a conventional computer. A micro makes up for this lack of resources with a small size, low power and low cost. A comparison of the characteristics can be seen in Table 1.1. A computer is typically defined as a multi-purpose, flexible unit able to do computation. A microcontroller on the other hand typically is hard-coded to do one specific job.

The terms *microcontroller* and *microprocessor* are different and should not be used interchangeably. A micro*processor* is an IC which is able to perform computation, but requires external memory and peripherals to function. A micro*controller* has the memory and peripherals built into it, allowing it to be fully independent. Furthermore, the interface in and out of a microprocessor is mainly just an address and data bus. In a microcontroller, these busses are internal to the device. The interfaces in and out of a microcontroller are configurable to be a wide variety of communication standards. This self-contained nature and ability to deal with a wide variert of signals allows a microcontroller to (as the name suggest) be embedded in a larger

	CPU	**RAM**	**Non-volatile**	**Power**	**Size/Mass**	**Cost**
Computer	Dual, 3 GHz	4 GiB	500 GB	100 W	Large	R 3000
Micro	48 MHz	8 KiB	32 KiB	50 mW	Small	R 15

Table 1.1: Comparison of specs of entry level computer to STM32F051C6.

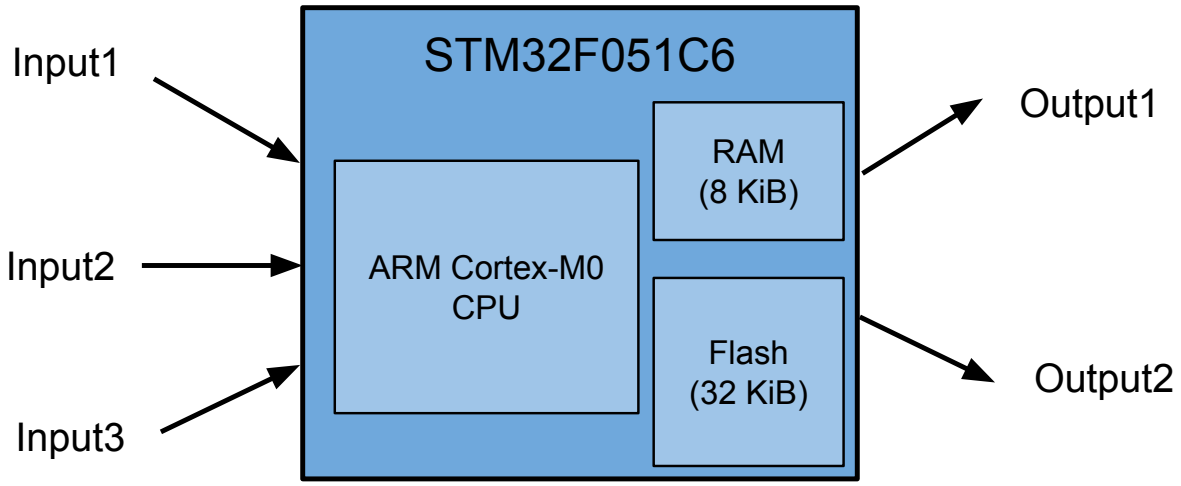


Figure 1.1: The most simplified view of the internals of the STM32F051

system and perform control and monitoring functions.

The micro we will be using is the STM32F051C6. It is manufactured by ST Microelectronics, but has an ARM Cortex-M0 CPU. ARM designed the CPU (specified how the transistors connect together). ST then takes this CPU design, adds it to their design for all of the other bits of the micro (flash, RAM, ports and much much more) and then produces the chip.

1.1.1 Development board block diagram

The development board consists of modules which connect to the microcontroller. Most of these modules are optional in that they are not required for the microcontroller to run. We will develop code later in the course to interface with some of these modules. Those which are not optional are the voltage regulator and the debugger. Following is a brief discussion of the purpose of each of the dev board modules (peripherals).

- STM32F051C6: This is the target microcontroller. It is connected to everything else on the board and it is where the code which we develop will execute.
- Debugger: this is essentially another microcontroller running special code on it which allows it to be able to pass information between a computer and the target microcontroller. The interface to the computer is a USB connection, and the interface to the target is a protocol called Serial Wire Debug (SWD) which is similar to JTAG. The specific type of debugger which we have is a ST-Link.
- Regulator: A MCP1702-33/T0 chip. This converts the 5 V provided by the USB port into 3.3 V suitable for running most of the circuitry on the board.
- LEDs: One byte of LEDs, active high connected to the lower byte of port B.
- Push buttons: Active low push buttons connected to the lower nibble of port A.
- Pots: 2 x 10K (or there abouts) potentiometers connected to PA5 and PA6.
- LCD Screen: A 16x2 screen connected to the micro in 4-bit mode. Used to display text.

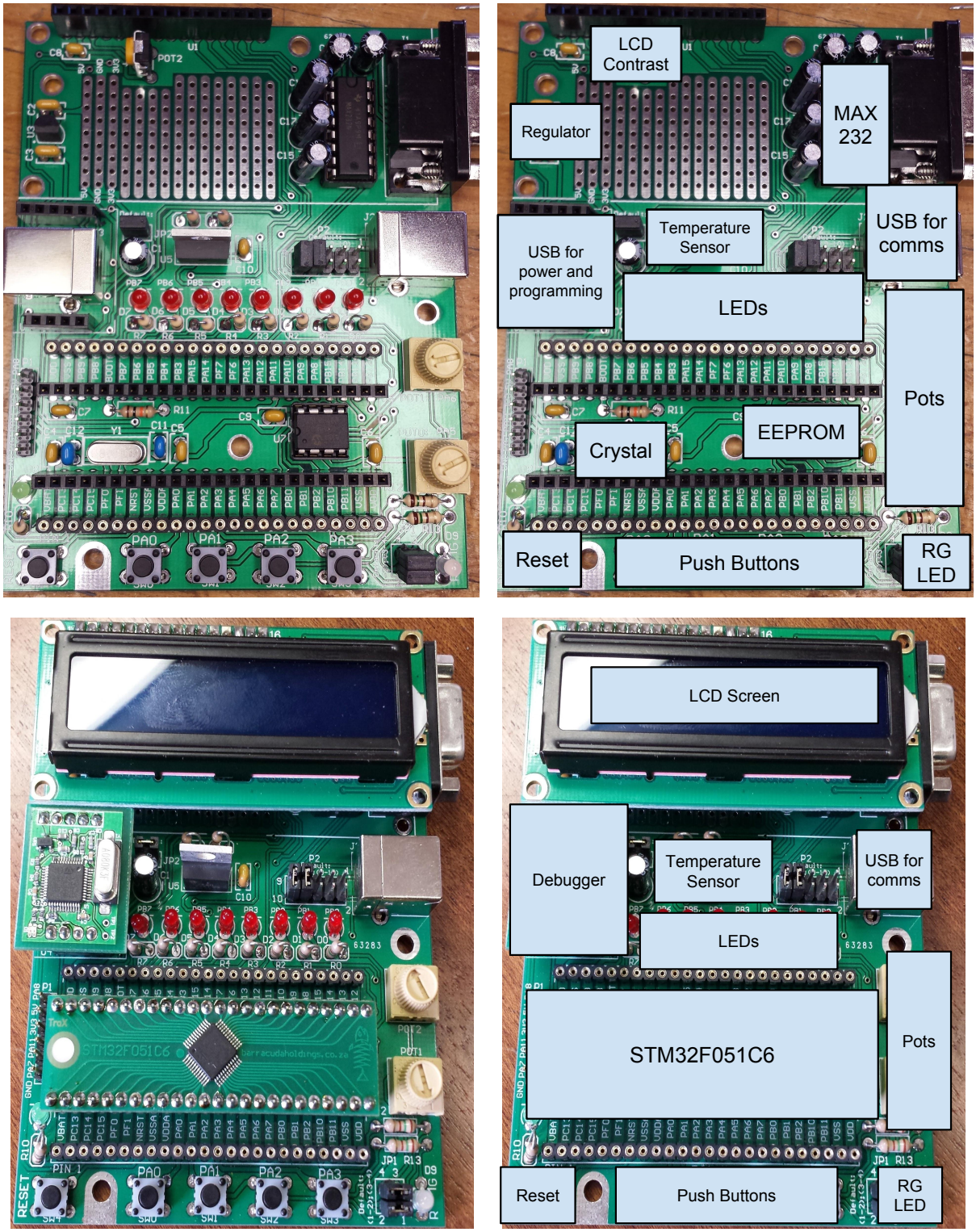


Figure 1.2: Modules on the dev board as seen when top boards unplugged or plugged in.

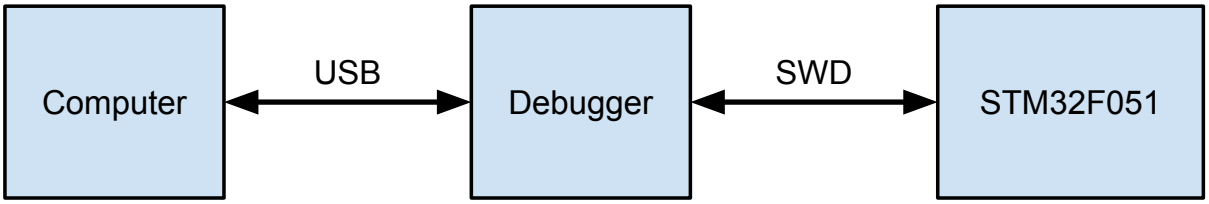
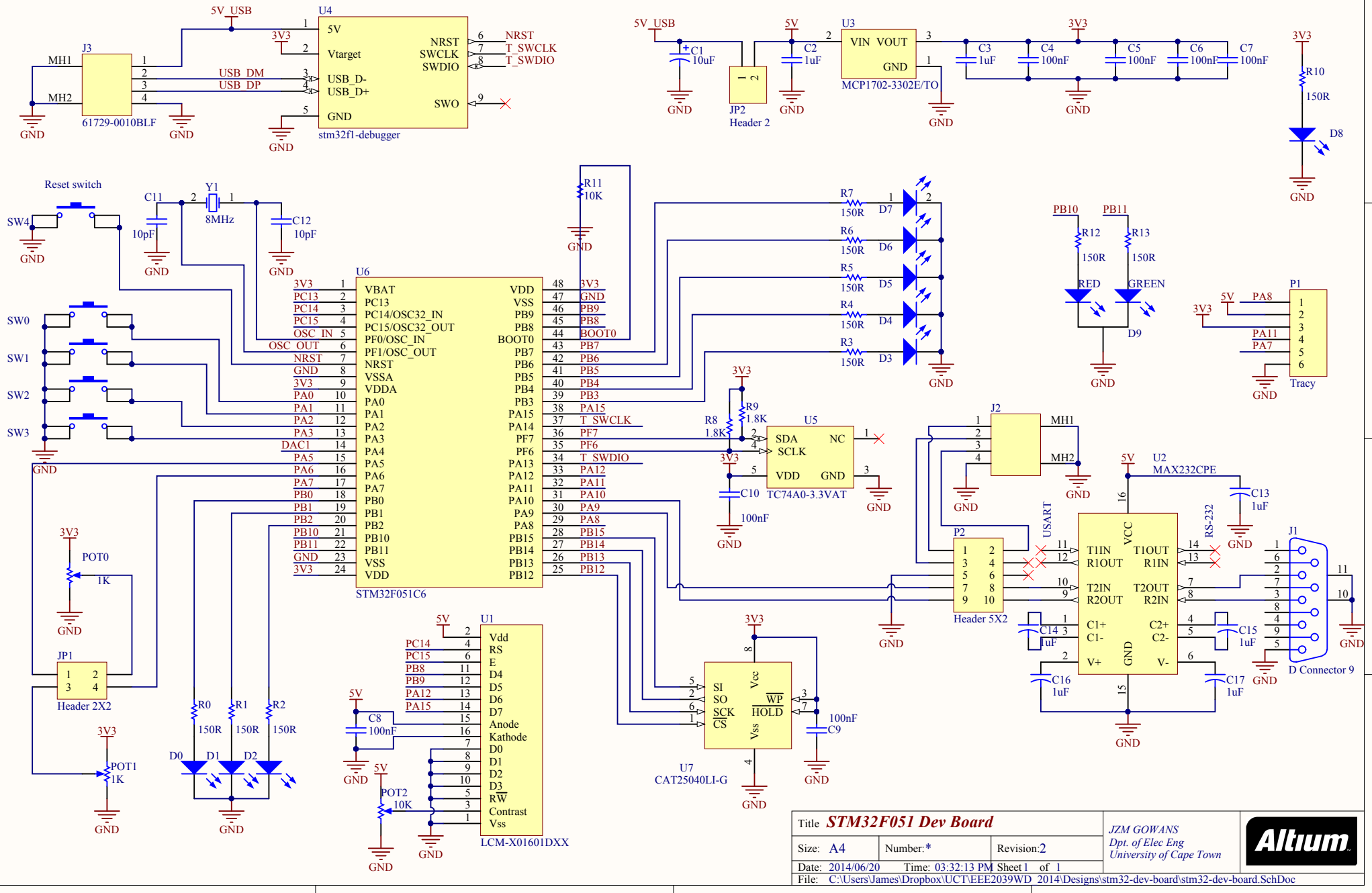


Figure 1.3: Highly simpified diagram showing how micro and computer communicate

- LCD contrast pot: The output of this potentiometer connects to the contrast pin of the LCD screen, hence allowing contrast adjustment.
- MAX232: This chips translates between TTL or CMOST logic level UART traffic and bi-polar higher voltage RS-232 traffic. Used for industrial communications links.
- USB for comms: The header allows intercepting of the UART traffic before it gets to the MAX232 and converting it to USB traffic through a small board which plugs into that header. When this facility is not being used, the jumpers on the header should be placed to allow the UART traffic to make its way to the MAX232.
- Temperature sensor: A TC74-A0 I^2C temperature sensor.
- Crystal: 8 MHz quartz oscillator with 10 pF caps for removing high frequency harmonics.
- EEPROM: A 25LC640A 64Kb Electronically Erasable and Programmable Read Only Memory (EEPROM) chip which communicates over SPI.
- RG LED: Common cathode Red/Green LED.

The full circuit schematic for the board follows. For now, we will forget about all of the other modules on the dev board and consider our system to be a computer talking to a debugger talking to a target micro, as shown in Figure 1.3. This is the most basic system which must be understood to allow us to load code onto the target microcontroller.



Title	***STM32F051 Dev Board***		
Size: A4	Number:*	Revision:2	
Date: 2014/06/20	Time: 03:32:13 PM	Sheet 1 of 1	
File: C:\Users\James\Dropbox\UCT\EEE2039WD 2014\Designs\stm32-dev-board\stm32-dev-board.SchDoc			

JZM GOWANS
Dpt. of Elec Eng
University of Cape Town

Altium

2 Memory Model

We will now beging to expand on some of the block is Figure 1.1. Before starting to explore how the CPU works, it's useful to have an understanding of how memory is laid out. We will start looking at the flash and RAM blocks. Together with another block called perihperals (which we will explore later), these blocks make up memory. It's important to note that this memory is located *outside* of the CPU, but still inside the microcontroller IC.

The memory of a device can be though of as a very long row of post boxes along a street. Each post box has an address, and each post box can have data put into it or taken out. The amount of data that each post box can hold is 8 bits, or one byte. Therefore, each memory address is said to address one byte. The address of each post box is 32 bits long, meaning that addresses range from 0 (0x00000000) to just over 4 billion (0xFFFFFFFF). In actual fact, the *vast* majority of these addresses do not have a post box at them. These addresses are said to be unimplemented. Only very small sections of this address space are implemented and can actually be read from or written to. Flash and RAM are contiguous blocks of memory, with a start address and an end address. A simplified memory map of the STM32F051 is shown in Figure 2.1. From this, we can see that if we want to use changeable variables in our programs, the variables should be located at addresses between 0x2000 0000 and 0x2000 1FFF. If we want to load code onto the micro which should not be lost when the device loses power, the code should be loaded into addresses between 0x0800 0000 and 0x0800 7FFF.

2.1 Data Types and Endianness

Very often we will need to work with clumps of data which are larger than 1 byte. ARM defines datatypes for a 32 bit CPU as follows:

- byte: 8 bits
- halfword: 16 bits
- word: 32 bits
- doubleword: 64 bits

Each memory address only addresses one byte of memory, so how can something like a word (four bytes) be stored in memory? Obviously, the four bytes have to come after each other to form a four byte block, or word. However, it is not obvious which order they should come in. For example, consider the case of wanting to store the word 0xAABBCCDD in address 0. The two possible ways of doing it are shown in Table 2.1. It doesn't really matter which one of these schemes is used - they each have their pros and cons and different processors use different methods. It is important to know which one our processor has chosen to use. Our processor uses little endian. A more abstract view of how data is stored in our processor is given in Figure 2.2

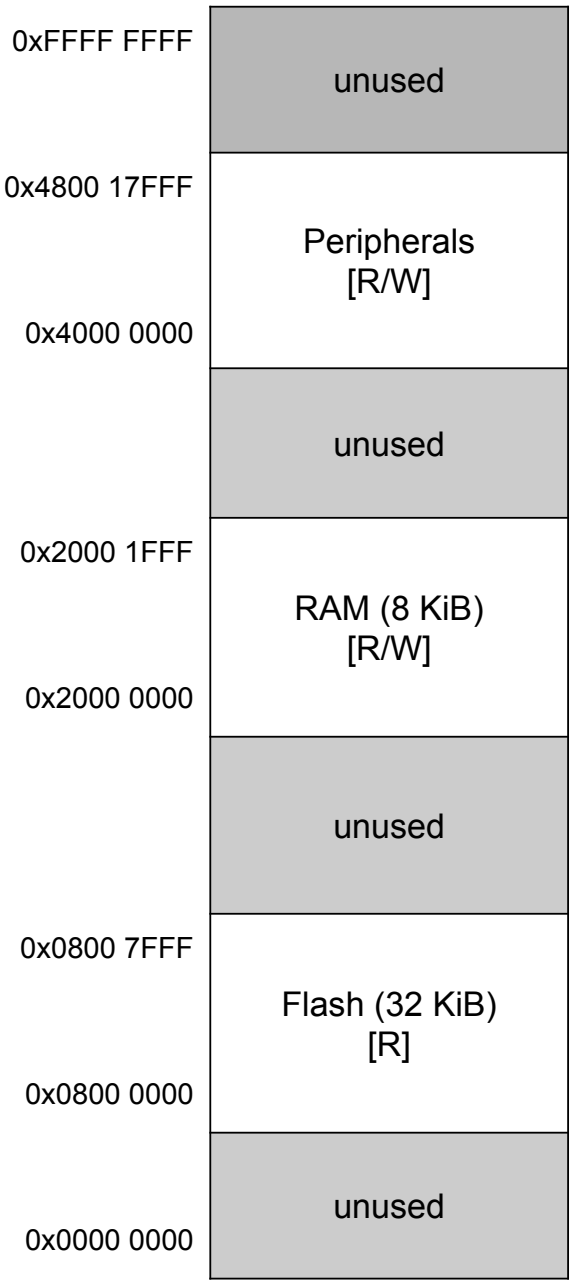


Figure 2.1: Simplified STM32F051C6 memory map. Note how all addresses are 32 bits. The blocks are very much not to scale. Source: datasheet, Figure 9

Little Endian		**Big Endian**	
Address	Data	Address	Data
3	0xAA	3	0xDD
2	0xBB	2	0xCC
1	0xCC	1	0xBB
0	0xDD	0	0xAA

Table 2.1: Layouts of the word 0xAABBCCDD in memory at effective address 0, according to little or big endian format

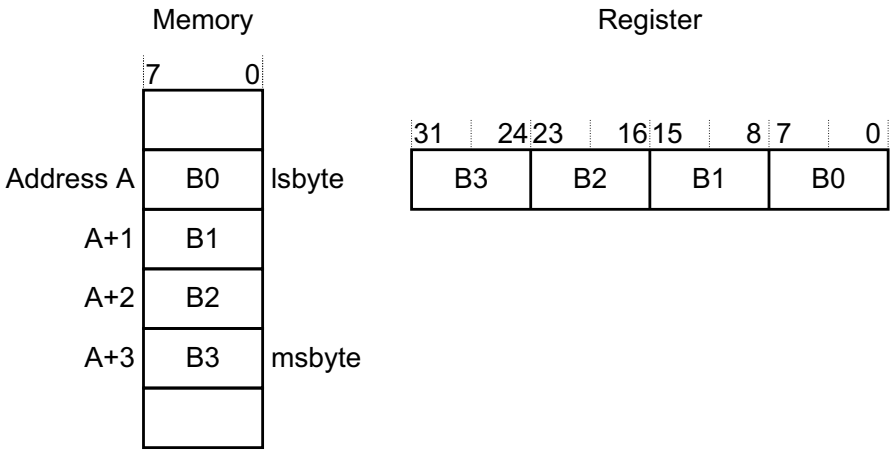


Figure 2.2: More abstract view of little endian layout. Source: Prog Man, page 28

3 The ARM Cortex-M0

At the core of a microcontroller is the CPU. Our CPU is called the Cortex-M0 and is designed by Advanced RISC Machines (ARM). The ARM Cortex-M0 CPU is certainly the most interesting block inside the STM32F051C6. This is where all processing happens, hence this is where the instructions which we write will run. It is therefore essential that we have an intricate understanding of the CPU so that we may write useful code for it. This chapter seeks to explore the CPU in some detail.

3.1 Programmer's Model of the CPU

A programmer's model is a representation of the inner workings of the CPU with sufficient detail to allows us to develop code for the CPU, but no unnecessary detail. The expanded view of the CPU which will now be discussed can be seen in Figure 3.1. This simple model of a CPU is a set of CPU registers, an Arithmetic and Logic Unit (ALU) and a control Unit. The CPU registers are blocks of storage each 32 bits wide which the CPU has the ability to operate on. Only data which is inside a CPU register can be operated on by the CPU. The ARM Cortex-M0 has 16 such registers.

The ALU is that which performs the operations on the registers. It can take data from registers as inputs, do very basic processing and store the result in CPU registers.

The control unit manages execution by telling the ALU what to do. Together, the registers, ALU and control are able to execute instructions. Examples of instructions which the CPU is able to execute:

1. adding the contents of R0 and R1 and storing the result in R6
2. copying the contents of R3 into R0
3. doing a logical XOR of the contents of R3 with the contents of R4 and storing the result in R3
4. moving the number 42 into R5

3.2 CPU Architecture

This section will explore some CPU architectures and compare them to the architecture of the Cortex-M0.

The Cortex-M0 makes use of a Von Neumann architecture. This means that there is a single bus which connects all of the parts (such as CPU, RAM, flash) inside the microcontroller. The implication of this is that the CPU cannot fetch an instruction from flash at the same time as it moves data in or out of RAM. This limitation allows for a much simpler architecture, but at the expense of performance.

Other microcontrollers (even others in the Cortex-M series like the Cortex-M3) follow a Harvard architecture, meaning that there are separate buses used for fetching instructions and

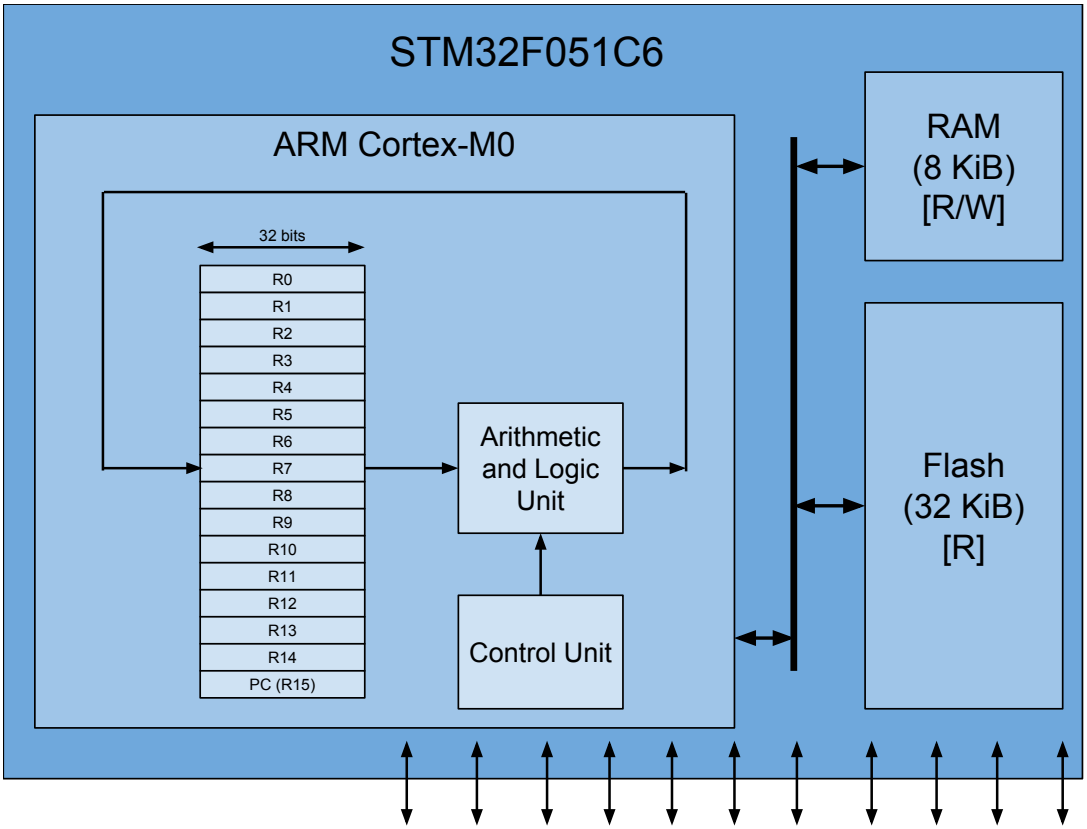


Figure 3.1: A view of the internals of the STM32F051 with the ARM Cortex-M0 expanded

moving data around. This allows faster execution as instructions can be fetched at the same time as data is loaded or stored. However, it necessitates greater complexity and more transistors.

It's been said that the ARM Cortex-M0 is a 32-bit processor. For comparison, the processor which we used in this course previously (MC9S08GT16A) was an 8-bit processor. Your personal computer probably has a 64-bit CPU. 16-bit CPUs are also quite common. So what exactly does it mean when we say that the processor is 32-bits? Essentially, the number of bits which a processor is said to be referrers to the size of the data bus. In other words: the amount of data which the processor is able to move around internally or perform arithmetic and logic operations on. Hence, with a 32-bit processor, we can move 32 bits of data from one spot in memory to another in just once instruction. If you had a 8-bit processor, it would cost 4 instructions to move 32 bits of data around.

3.3 Program Counter

The Program Counter is a special register in the CPU, specifically: R15. It's called "special" because it has a specific, fixed purpose and cannot be used as a general purpose register like the other registers can. It's purpose is keeping track of were we are in the execution of a program. All instructions which need to be executed are laid out sequentially in flash, each instruction occupying a halfword of memory. Hence, each instruction has a defined address. The PC points to (ie: hold the address of) the instruction which is about to be fetched from flash and executed.

Typically, the value of the PC is simply incremented by 2 in order to cause it to point to the next instruction in memory. However, it's possible to alter the flow of execution of a program by issuing a *branch* instruction which will cause the PC to be incremented or decremented by a different amount.

3.3.1 Three stage pipeline

There is a bit more complexity to the program counter than initially apparent. It's worth understanding this extra intricacy as it affects how other instructions which depend on the program counter work. The ARM Cortex-M0 implements a three stage pipeline. This means that an instruction is broken up into three parts, and executed over the course of three clock cycles. The parts are:

- **fetch:** the instruction which the program counter points to is pulled into the CPU.
- **decode:** the CPU control unit "looks" at the 16 bits which represent the instruction, and figures out what action it must take.
- **execute:** the CPU runs the instruction, causing data to be modified.

The fact that the CPU is pipelined means that different instructions can be going through different phases *at the same time*. In other words, one instruction can be being fetched while another is being decoded while another is being executed. As an example, assume we have three instructions which we want to execute, instruction A, instruction B and instruction C. The three instructions being run through the pipeline is shown graphically in Figure 3.2. It's critical to note how the program counter is always pointing to the instruction being *fetched*. This makes sense as the job of the program counter after all is to facilitate keeping track of which instruction must be fetched. For this reason, when an instruction is being executed, the PC is actually pointing to two instructions (four bytes) further ahead in memory, and *not* at the address of the instruction in execution. Hence, when an instruction in execution uses the PC, the value which will be used is the address of the instruction plus four.

3.4 Reset Vector

Initial address of PC loaded from 0x0800 0004.

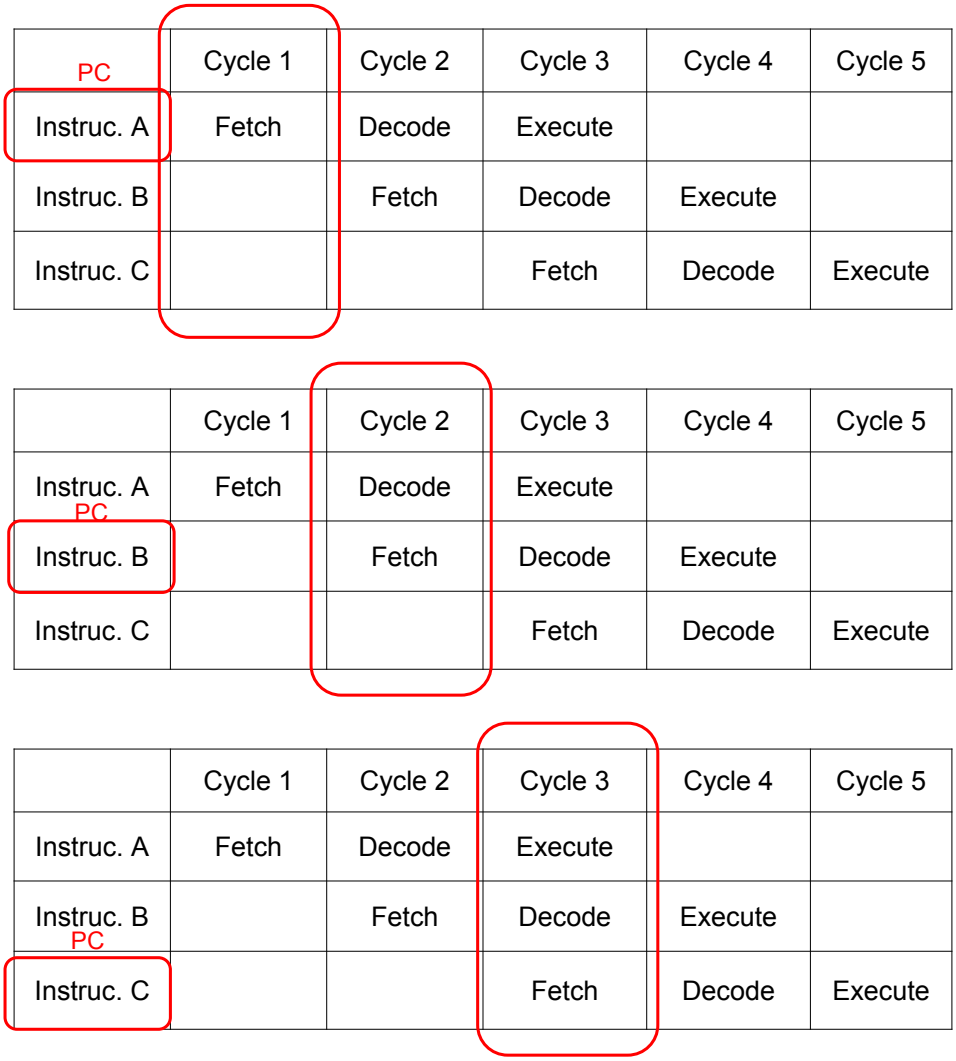


Figure 3.2: Showing three instructions being run through a three stage pipeline, as well as where the PC is pointing every cycle

4 Coding

4.1 Assembly

In order to get the CPU to do some of what we've discussed above, it needs to have code loaded onto it to run. We write code in a language called assembly. Assembly is a human-readable language. A program is made up of a sequence of instruction; each instruction gets executed by the CPU. It's quite easy to see what each instruction does by reading the program. The complete instruction set is located in the Programming Manual. You must be familiar with this document! Examples of instruction which carry out the tasks listed above are:

1. `ADDS R6, R0, R1`
2. `MOV R0, R3`
3. `EORS R3, R3, R4`
4. `MOVS R5, #42`

The CPU does not have the ability to understand our nice English words like *ADD* or *MOV*. The CPU only has the ability to understand binary data. Assembly code must be compiled to machine code. A machine code instruction is a binary string, 16 bits long consisting of the operation code (opcode) and the data which it must operate on (operand). For example, assume that we wanted to ascertain the machine code representation of the instruction `ADDS R6, R0, R1`. An extact from the ARMv6-M Architecture Reference Manual is shown in Figure 4.1 where Rd is the destination register and Rm and Rn are the source registers of the add. It can easily be seen that the instruction would compile to `0001100 001 000 110 = 0x1846`. The opcodes for each instruction are detailed in the ARMv6-M Architecture Reference Manual. All of the instructions in the program are 16 bits long and are stored sequentially after one another in flash memory.

4.1.1 Instruction Sets

An instruction set is the collection of all of the instructions which a processor can execute. The ARM Cortex-M0 uses the ARMv6-M architecture and this architecture supports the Thumb instruction set (as opposed to Thumb-2 or ARM). Thumb contains about *XXX* instructions,

Encoding T1 All versions of the Thumb instruction set.

`ADDS <Rd>,<Rn>,<Rm>`

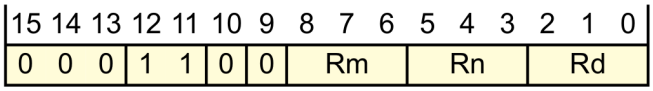


Figure 4.1: An encoding of the ADDS instruction

each of which is 16 bits long.

Higher end ARM processors such as the Cortex-M3 or Cortex-M4 support the ARMv7-M architecture which allows multiple instruction sets to be supported by the processor. The ability to support multiple instruction sets requires *interworking*. Interworking is the ability to specify to the CPU which instruction set to use. While our ARM Cortex-M0 only supports the Thumb instruction set, there is no need for interworking, yet the cabability has still been incorperated into the architecture to allow for compatability to other processors. This means that although our processor only supports one instruction set (Thumb), we have to explicity tell it that we are using that instruction set.

4.2 Linking

Once our assembly code has been written and compiled to machine code, the computer which loads the code onto the micro has to be told what addresses to place the code at. The code should be placed starting at the beginning of flash.

4.2.1 Executing Code

The PC always points to the instruction which is about to be fetched. Hence, when your micro boots up, before it has executed anything, the PC will point to the first instruction to be fetched/decoded/executed. By "point to" we mean that it holds the address of the instruction.

As each instruction in the ARM Cortex-M0 instruction set it 16 bits (aka: half a word) long, ARM have implemented a rule that all instructions must be half word alligned. In other words, the address of the instruction must be divisible by 2 bytes. Legal addresses for instructions are hence, 0x02, 0x04, 0x06, 0x08 ... etc. This means that the least significant bit (bit 0) of the PC register is unused in specifcying the address of an instruction. Hence, it has been assigned another use. Specifically, to indicate the instruction set which is being executed.

5 Loading and Storing

Loading is the process of getting data from somewhere in the memory space into the CPU registers so that it can be used in processing. Storing is the process of getting data which is in the CPU registers into memory. Remember that seeing as flash is read-only memory, we cannot store data to flash address, but we can store to RAM.

The general format for a load is that a destination register, a register containing a base address, and an offset are supplied. An effective address is then calculated as the base address plus the offset. The contents of memory at the effective address are then copied from memory into the destination CPU register.

A store operation is very similar. Again, a register containing a base address and an offset are supplied, but this time it is a source register not a destination register which is supplied. Again, and effective address of base plus offset is calculated. The contents of the source register is copied into the effective address.

Note that most of the load/store operations which we will be doing are 32-bit (word) load or stores. This is because the CPU registers are 32 bits. So far we have only spoken of a single effective address. As you know, each address can only hold 8 bits. Hence, in order to load or store 32 bits, four sequential addresses are used. The effective address specifies the *lowest* in the sequence of the addresses. For example, if we wanted to store the contents of R0 in 0x20000000, the word would be placed into the address range 0x20000000, 0x200000001, 0x20000002 and 0x20000003. Remember that our processor uses little endian format, so the LSB is placed at 0x20000000 and the MSB at 0x20000003.

We will now explore some implementations of loading and storing.

5.1 Immediate Offset Loading

In this format, the base address is supplied in one of high CPU registers (R0 - R7), and the offset is supplied as an immediate number. The instruction format for loading data into a register is

```
LDR Rt, [Rn, #imm]
```

where `Rt` is the target register for the load, `Rn` contains the base address and `#imm` is the offset from the base address.

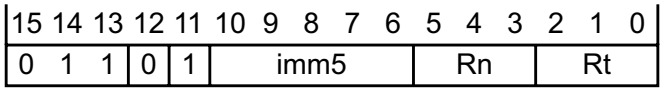
The way that this instruction works is that it calculates an *effective address* which is equal to the contents of the base address register plus whatever number is supplied as an immediate operand. There is, however, a slight complexity in how the offset is dealt with.

5.1.1 Offset restrictions

Remember that all instructions are limited to 16 bits. The format of the LDR instruction in machine code is shown in Figure 5.1. We can see that after 5 bits of opcode and $2 \times 3 = 6$ bits of register specifications, we are only left with 5 bits of offset. Normally, these 5 bits would only allow us to provide an offset of $2^5 - 1 = 31$ bytes. This is not very much! In order to extend the range of the 5 offset bits, the actual offset used is equal to the 5 bit immediate number multiplied

Encoding T1 All versions of the Thumb instruction set.

```
LDR <Rt>, [<Rn>{,#<imm5>}]
```




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
```

Figure 5.1: Machine Code representation of LDR instruction. Source: ARMv6-M Architecture Reference Manual

by four. This multiplication by four is the same as appending two zeros to the end of the binary value, which you can see is being done in Figure 5.1. This means that the amount which we are able to offset a base address by is now $(2^5 - 1) \times 4 = 124$, which is significantly more useful. However, seeing as we are multiplying to immediate number by four to get the actual offset, the implication is that all offsets *must* be a multiple of four. The compiler automatically takes care of dividing whatever offset we supply in our assembly instruction by four in order to get it to fit into the 5 bit immediate number, and the CPU then multiplies the immediate number by four to get the offset.

For example: if we wanted an offset of 12, the immediate number which would be placed in the instruction by the compiler would be 3.

5.2 Program Counter Relative Loading

There is another format of the LDR instruction which takes the Program Counter as a base register, and allows for an 8-bit immediate offset. If you wish to load data from flash into a CPU register, it makes sense to use the PC as a base register due to the fact that the PC is already initialised to be pointing to an address in flash. Specifically, it is pointing to the instruction which is being fetched (not executed - remember the three stage pipeline!). The format of the LDR instruction for PC relative loading can either be specified in the same was as the general LDR instruction, or it can have a label provided as an operand, as follows:

```
LDR Rt, [PC, #imm]
LDR Rt, <label>
```

If one supplies a label as an operand, all that the compiler does is calculate the correct immediate offset value to insert, and compiles the instruction as if it were in the first format. It's important to note that these instructions are exactly equivalent: all that using a label does is cause the compiler to do the hard work of calculating the correct offset so you don't have to. It would really be a lot of hard work; every time you changed something in the structure of your program which caused instructions to be moved to different memory addresses (link writing a new line of code!) you'd potentially have to re-calculate your offsets. The ability to use labels is one of the most useful features of the compiler.

5.3 Register Offset Loading

So far all offsets have been supplied as immediate numbers to the load instructions. However, there is another format of the load instruction called a register-offset load. Here, the offset is contained in another register. This is useful as the offset can be set at run-time by modifying the contents of a register, rather than at compile time. In this case, the effective address is calculated as the contents of the base register (`Rn`) plus the contents of the offset register (`Rm`).

LDR Rt, [Rn, Rm]

5.4 Storing

The storing commands are so similar to the loading that they will barely be discussed. One difference is that there is no PC-relative store, as there would be no point trying to store data to read-only memory. The store instruction takes moves the contents of a source register, `Rt`, and places it at the effective memory address equal to the base address, `Rn`, plus an offset either supplied as a 5-bit immediate number, `#imm5`, or in an offset register, `Rm`.

STR Rt, [Rn, #imm5]
STR Rt, [Rn, Rm]

6 Perihperals

Peripherals in our context can have two meanings. Either, they could be the devices around the microcontroller on the development board like the LCD, push-buttons, potentiometers, temperature sensor or EEPROM which the microcontroller is able to interact with, usually for the purpose of getting input, displaying output or storing data. Alternatively, peripherals could refer to the blocks of circuitry inside of the microcontroller which provides some additional functionality which the CPU does not have. Examples would include circuitry for providing precise timing, or circuitry to interact with the pins of the microcontroller. To distinguish between the two, we call those perihperals which are outside of the microcontroller development board peripherals, and those which are inside the microcontroller chip we call internal peripherals.

The general structure is that the CPU interfaces with internal peripherals which in turn interface with dev board peripherals through the pins on the microcontroller.

6.1 Internal Peripherals

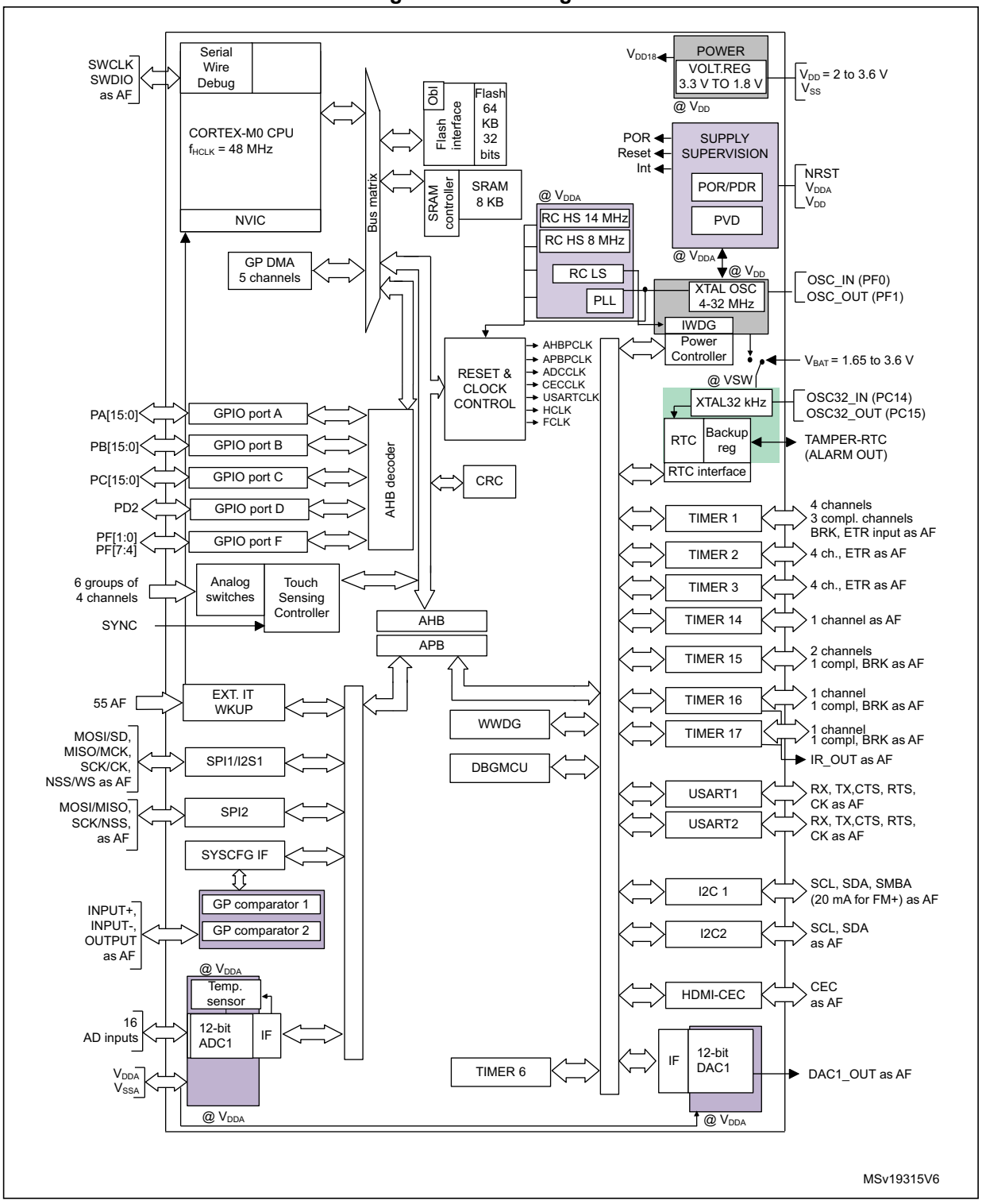
All internal peripherals are organised in a bus architecture which allows the CPU to interact with them. The full diagram of all of the peripherals in the STM32F051 is shown on the following page. In order for the CPU to interface with them, each peripheral has a block of memory associated to it. Recall the address space of the microcontroller as shown in Figure 2.1. The block called *peripherals* running from address 0x4000 0000 to 0x4800 17FF is the range of addresses which is available to have peripherals associated with it. The full memory map can be seen in Figure 2 of the Reference Manual.

Out of that large peripherals block of memory, each peripheral has a specific block of memory associated with it. The starting and ending address for each peripheral in the microcontroller can be seen in Table 1 of the Reference Manual. Note how the vast majority of the peripherals address space is unimplemented (or "reserved"). This allows there to be lots of space for expansion: fancier micrcontrollers can have more peripherals and make use of this unimplemented address space.

Inside each block of memory assigned to a specific peripheral is further sub-divisions of the block into *registers*. Registers are blocks of memory (typically one word big on our processor) which provide a specific, well defined element of functionality, typically configuring how the peripherals works or providing some status information about the peripheral. The CPU is able to write data to a register to configure the peripheral or read data from a register to get information about the peripheral. Sometimes a register simply holds a number (for example: for use in a counter) but more frequently each individual bit in a register as a specifc meaning. For example, a bit can be set high to enable some sort of functionality or set low to disable some functionality.

Each register has an address which must be known when interacting with that register. The way that the address is calculated is using a (base address) plus (offset) system. The base address is the start of the address range for the peripheral as seen in Table 1 of the Reference Manual, and the offset is the number which must be address to the base address to get the effective

Figure 1. Block diagram



address of the register. This is a very convenient system as our load and store operations in the CPU also work on a base plus offset system.

A description of what each register does (and indeed what each bit in the register does) as well as the offset for that specific register can be found at the end of the chapter of the Reference Manual which deals with the peripheral (or class of peripherals) which you're trying to interact with.

A register is like RAM in that it is volatile memory, but it is different to RAM in that while RAM is general purpose memory which can be used for storing whatever you like, each register has a specific function and very specific, meaningful data must be written to or read from the peripheral which will configure the microcontroller in some way.

The following chapters serve to describe the operation of some of the key peripherals in the microcontroller.

7 General Purpose Input/Outputs

TODO: Pins and ports

One of the simplest ways to interface the microcontroller with external circuitry is via GPIO. Most pins on the microcontroller are able to operate in GPIO mode. As the name implies, a GPIO pin can be either an input or an output. Additionally, a pin can be placed into an alternate function or analogue mode; these will be discussed later.

A diagram showing how the pin is structured electrically inside the microcontroller is shown in Figure 7.1. The ability for the microcontroller to communicate with external devices via GPIO pins is one of the defining differences between microcontrollers and microprocessors.

7.1 Pin Mode

As mentioned, the pin can be in one of four possible modes: input, output, alternate function, analogue. There is a register which controlled which mode the pin operates in, known as the GPIOx_MODER. The 32 bits of the register are divided up into pairs of bits where each pair of pits sets the mode for the associated pin.

7.1.1 Input Mode

Input mode is the default mode for most pins. In this mode, the pin is measuring the voltage applied to it and ascertaining whether it is a logic 0 or a logic 1. This 'decision' is made by a Schmitt Trigger which has useful things such as well defined high and low levels, hysteresis and high impedance. The logic level of each pin is latched on each clock cycle and written to the Input Data Register (GPIOx_IDR). As each pin can only be considered to be either a logic high or a logic low, there is only 1 bit necessary to represent the state of a pin.

7.1.2 Output Mode

Here, the pin does not measure a logic level, but rather asserts a logic level. When in output mode, the pin will either assert a logic 0 allowing it to sink current from an external source, or assert a logic 1 allowing it to source current into an external sink. The logic level which is asserted is controlled by the Output Data Register (GPIOx_ODR).

How to set or clear individual bits

There is often a case where you wish to modify only one or two of the bits of a port, leaving the rest of the pins unchanged. If you simply write a pre-defined value to the pins, it will force *all* of them to take on a specific value. The way to modify only a single bit is to do a logic AND or OR of the contents of the register with a pre-defined pattern. An OR has the ability to set specific bits while leaving others unchanged, while and AND has the ability to clear certain bits while leaving the others unchanged. For example, say we wanted to set bits 1 and 2, while clearing bits 0, 3, 4 and 5, leaving the other bits of the port unchanged. We could do something like the following:

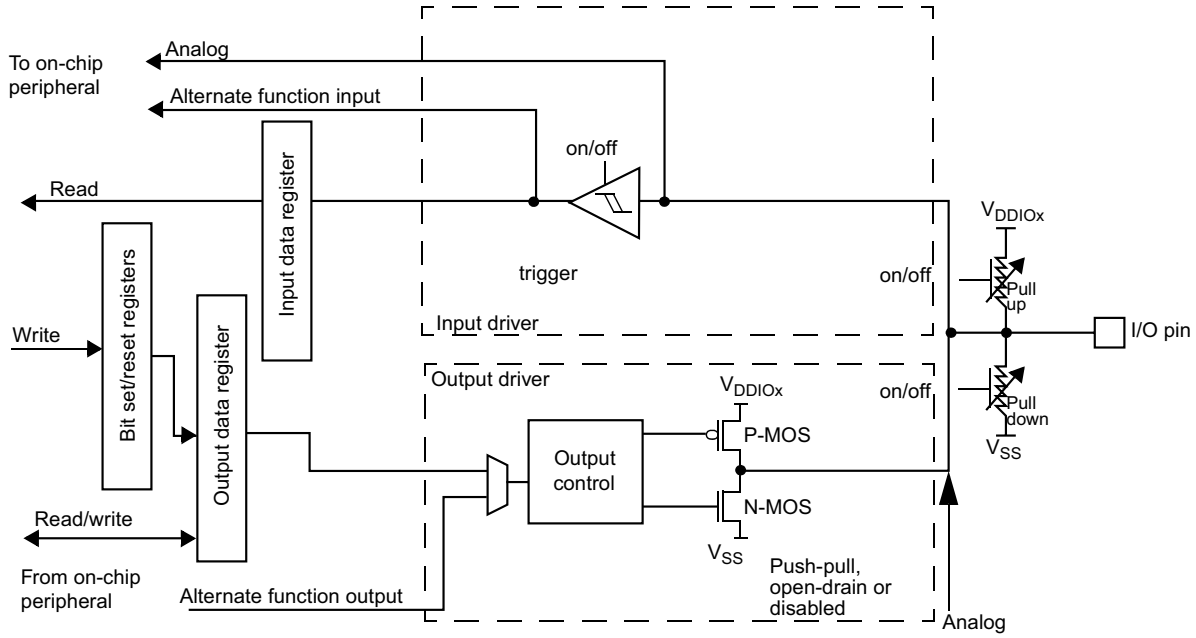


Figure 7.1: Internal structure of pin. Source: Figure 17, Reference Manual

```
@ assuming Rn contains the address of the register to modify:
LDR R0, [Rn]
LDR R1, =0b11111111111111111111111111000110
ANDS R0, R0, R1
MOVS R1, #0x06
ORRS R0, R0, R1
STR Rt, [Rn]
```

Each bit in this register can be set by writing to the register. Additionally, the bits in this register can be set via the Bit Set and Reset Register (GPIOx_BSSR). This register allows atomic (done in a single instruction) setting or clearing of individual bits in the ODR.