

# Introduction to Microcontrollers Notes

James Gowans

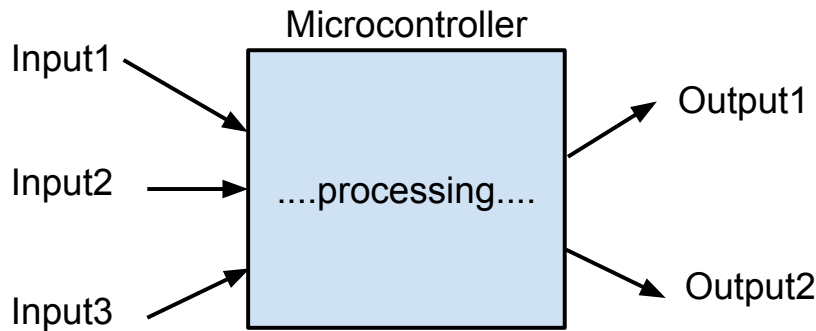
July 23, 2014

## **Licence**

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

## Contents

<b>1</b>	<b>Week 1</b>	<b>3</b>
1.1	What is a microcontroller? . . . . .	3
1.2	Development board block diagram . . . . .	3
1.3	Programmer's Model of the CPU . . . . .	5
1.4	Memory Model . . . . .	6
1.5	Writing and compiling code . . . . .	6
1.6	Instruction Sets . . . . .	8
1.7	Executing Code . . . . .	8
1.8	A basic model of the STM32F051 . . . . .	8
1.9	The ARM Cortex-M0 . . . . .	8
1.10	A Short History of ARM . . . . .	9



## 1 Week 1

### 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:

- CPU: Capable of doing arithmetic and logic operations. Operate on data. Execute instructions.
- Non-volatile memory (Flash): Holding information which must not be lost: program code.
- Volatile memory (RAM): Holding information which must be able to change, like program variables.
- Ports: Ways to get signals in and out of the system

However, 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 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 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.

### 1.2 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.

The debugger is essentially another microcontroller which is able to pass information between a computer and a 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

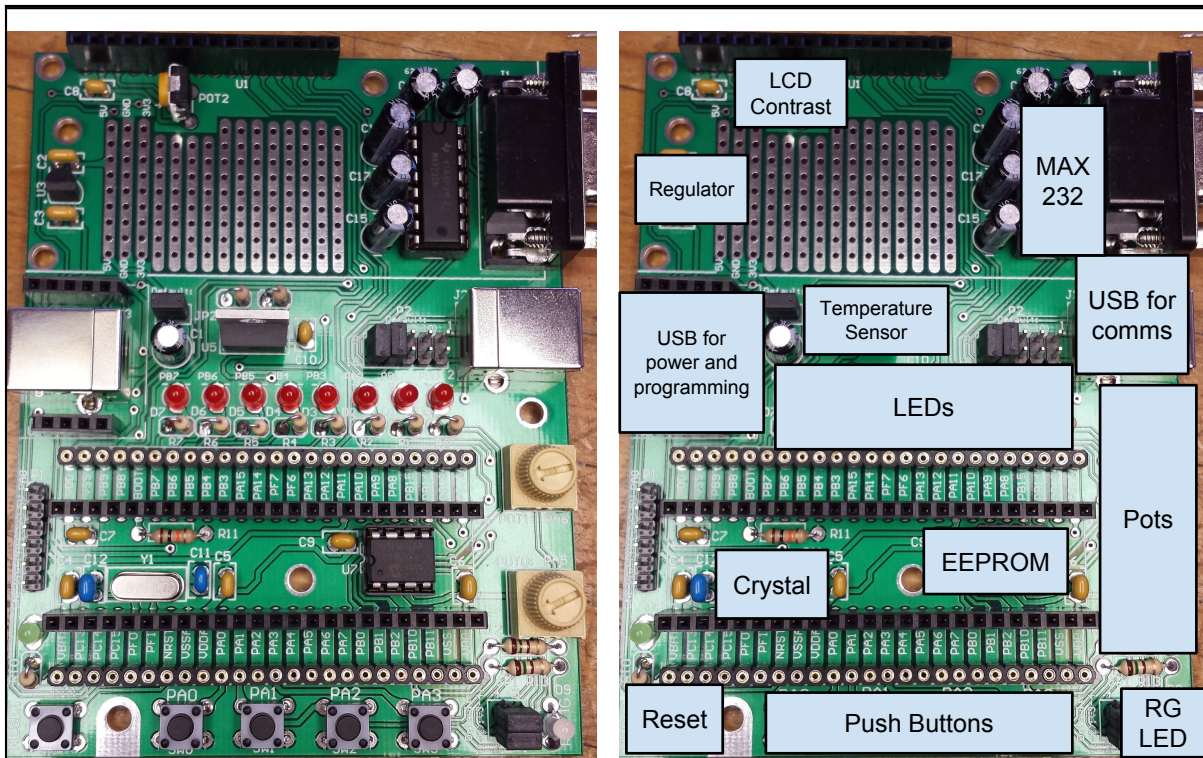


Figure 1: Modules on the dev board as seen when top boards unplugged

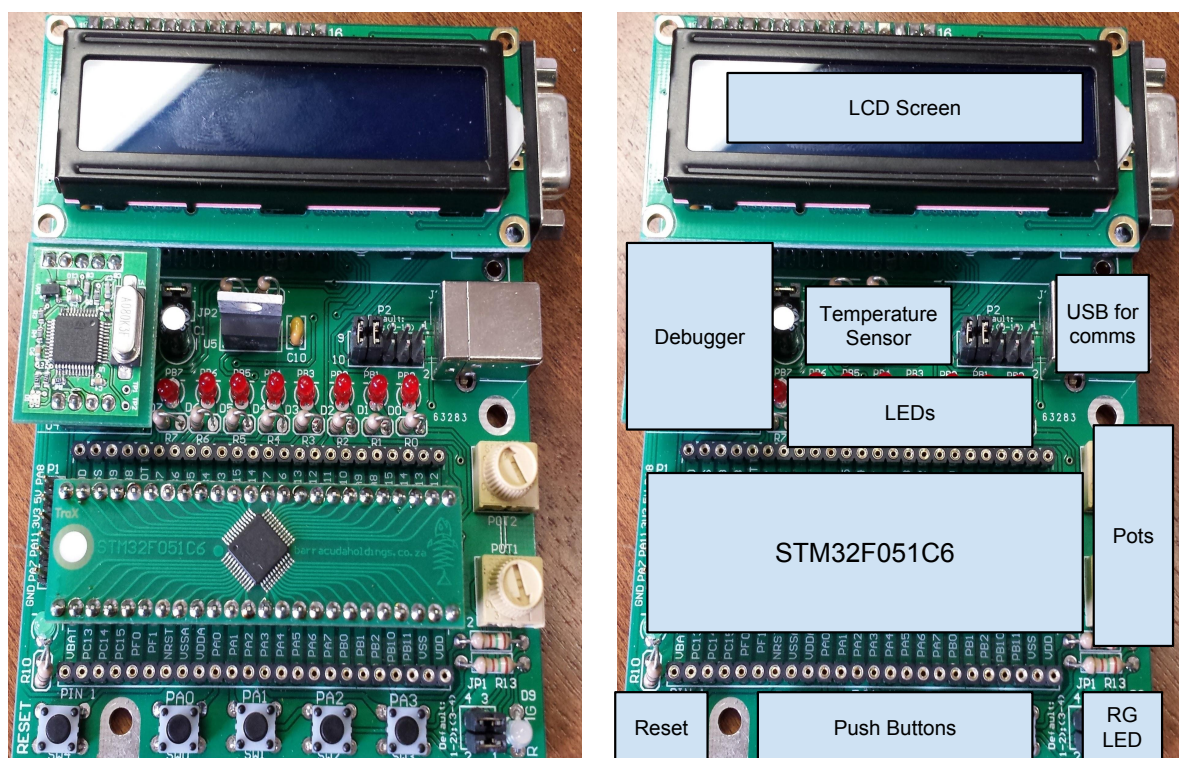


Figure 2: Modules on the dev board as seen when top boards plugged in

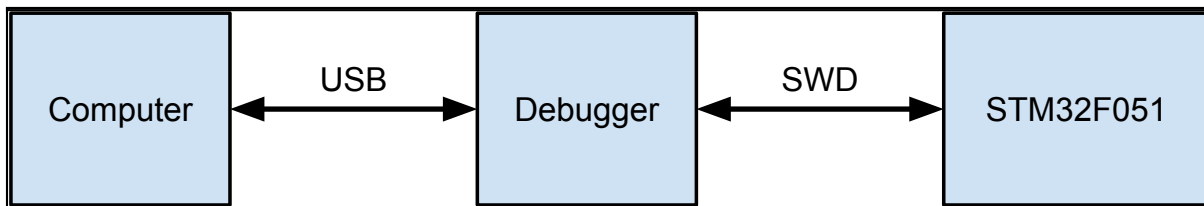


Figure 3: Highly simplified diagram showing how micro and computer communicate

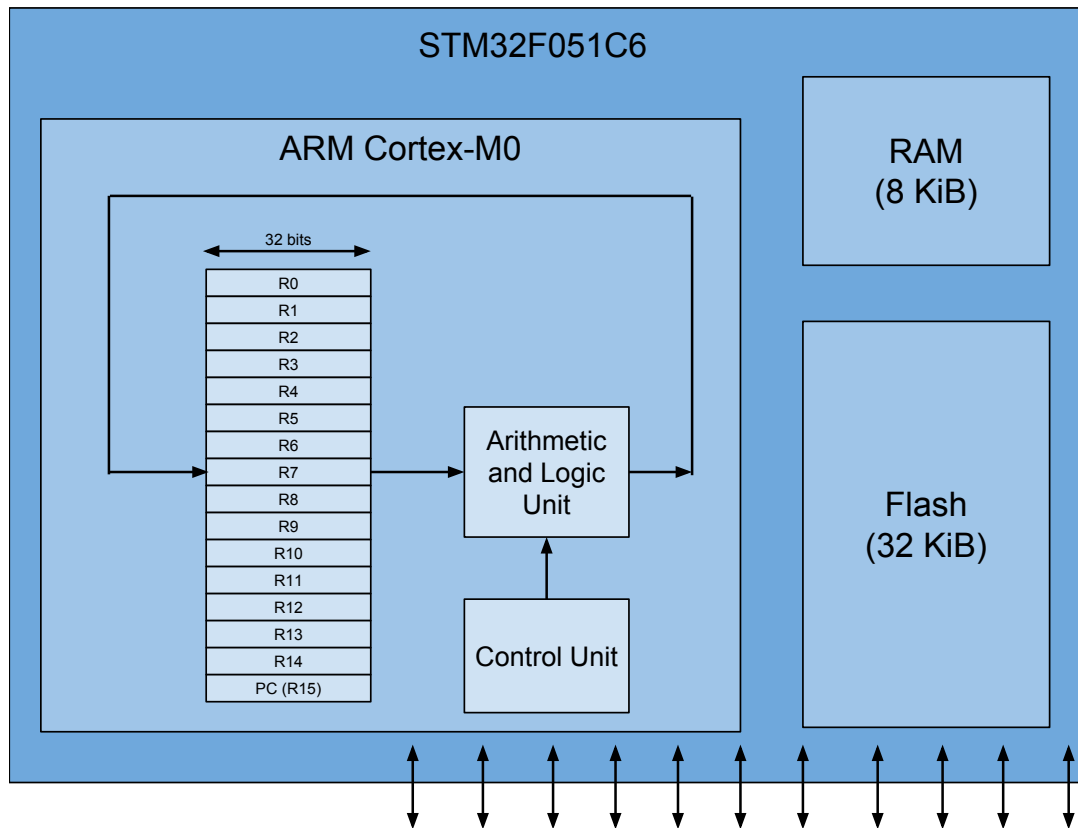


Figure 4: The most simplified view of the internals of the STM32F051

JTAG. 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 3.

Our attention will now turn to just the target micro.

### 1.3 Programmer's Model of the CPU

A 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. The ARM Cortex-M0 has 16 such registers. The ALU is that 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:

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

ARM defines datatypes for a 32 bit CPU as follows:

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

## 1.4 Memory Model

The memory of a device can be thought 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. 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. The sections which we are interested in are flash, RAM and peripherals (more on these later). Flash and RAM are contiguous blocks of memory, with a start address and an end address.

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.

## 1.5 Writing and compiling code

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 instructions; 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 instructions which carry out the tasks listed above are:

1. ADDS R6, R0, R1
2. MOV R0, R3
3. EORS R3, R3, R4
4. MOV 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. Our nice human-readable assembly code must be compiled to machine code. Machine code is a binary string, 16 bits long consisting

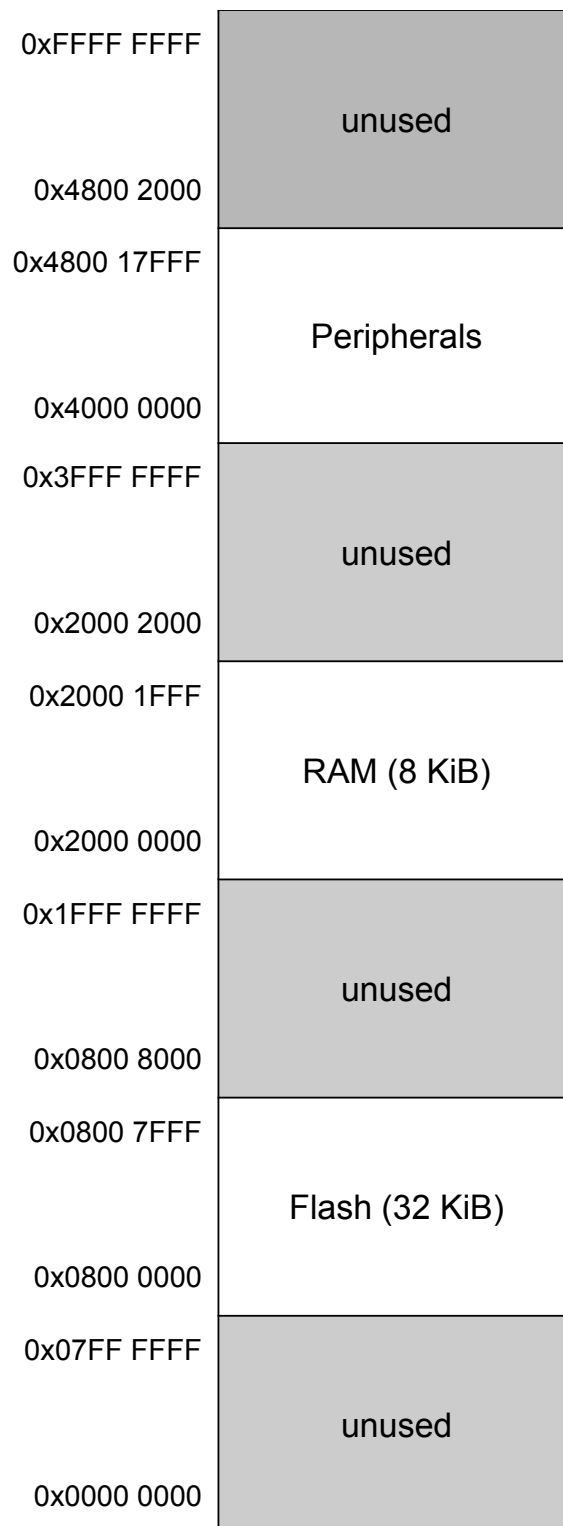


Figure 5: Simplified STM32F051C6 memory map. Note how all addresses are 32 bits. The blocks are very much not to scale. Source: datasheet, Figure 9

of the operation code (opcode) and the data which it must operate on (operand). For example, *ADDS R6, R0, R1* compiles to *(opcode for ADDS)(R1)(R0)(R6) = 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.

## 1.6 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, 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 capability has still been incorporated into the architecture to allow for compatibility to other processors. This means that although our processor only supports one instruction set (Thumb), we have to explicitly tell it that we are using that instruction set.

## 1.7 Executing Code

Once the code has been loaded onto the microcontroller, it will execute one instruction after the next. CPU register R15 is reserved for keeping track of where the micro is in execution. It is known as the Program Counter (PC). The PC always points to the instruction which is ABOUT to be executed. Hence, when your micro boots up, before it has executed anything, the PC will point to the first instruction to be executed. By “point to” we mean that it holds the address of the instruction.

As each instruction in the ARM Cortex-M0 instruction set is 16 bits (aka: half a word) long, ARM have implemented a rule that all instructions must be half word aligned. 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 specifying the address of an instruction. Hence, it has been assigned another use. Specifically, to indicate the type of instruction which is being executed.

## 1.8 A basic model of the STM32F051

## 1.9 The ARM Cortex-M0

The microcontroller which we will be using is the STM32F051C6. At the core of this micro is its CPU, which is called the Cortex-M0 and is designed by Advanced RISC Machines (ARM).a]]

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 refers 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.

## **1.10 A Short History of ARM**

Acorn