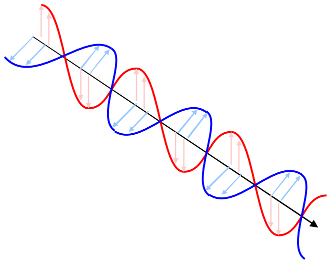
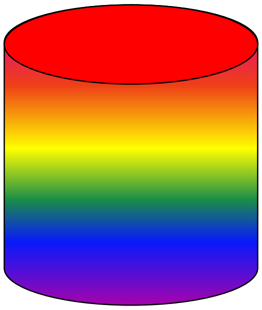
Microcontrollers

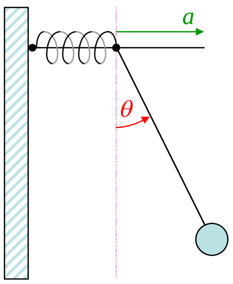
A Simple Introduction

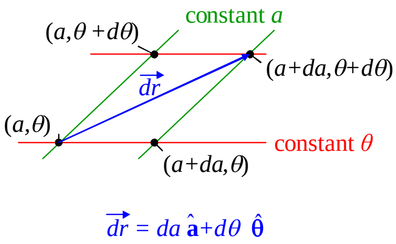
A Work In Progress. See physics.ucsd.edu/~emichels for the latest versions of the Funky Series.

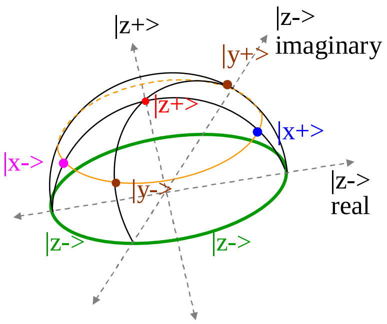
Eric L. Michelsen

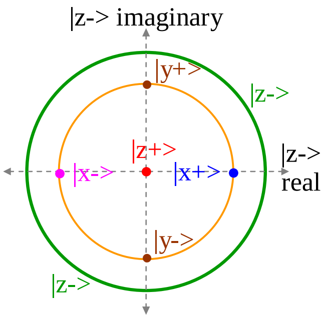


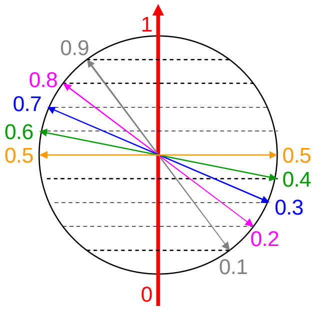


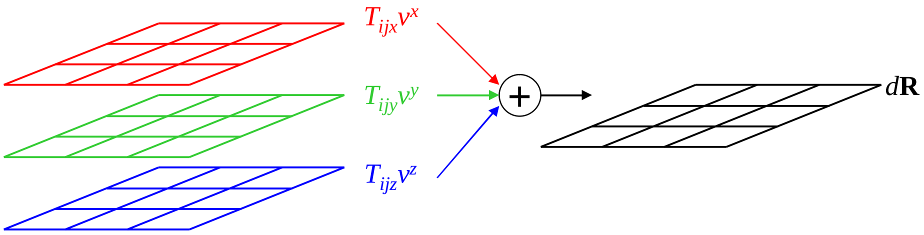












The above graphics have nothing to do with microcontrollers.

2006 values from NIST. For more physical constants, see http://physics.nist.gov/cuu/Constants/ .

Speed of light in vacuum	c = 299 792 458 m s^{-1} (exact)
Gravitational constant	G = 6.674 28(67) x 10–11 m3 kg–1 s–2
Relative standard uncertainty	±1.0 x 10^{-4}
Boltzmann constant	k = 1.380 6504(24) x 10^{-23} J K^{-1}
Stefan-Boltzmann constant	σ = 5.670 400(40) x 10^{-8} W m^{-2} K^{-4}
Relative standard uncertainty	±7.0 x 10^{-6}
Avogadro constant	N_A, L = 6.022 141 79(30) x 10^{23} mol^{-1}
Relative standard uncertainty	±5.0 x 10^{-8}
Molar gas constant	R = 8.314 472(15) J mol^{-1} K^{-1}
calorie	4.184 J (exact)
Electron mass	m_e = 9.109 382 15(45) x 10^{-31} kg
Proton mass	m_p = 1.672 621 637(83) x 10^{-27} kg
Proton/electron mass ratio	m_p/m_e = 1836.152 672 47(80)
Atomic mass unit (amu)	1.660 538 86 × 10^{-27} kg
Elementary charge	e = 1.602 176 487(40) x 10^{-19} C
Electron g-factor	g_e = –2.002 319 304 3622(15)
Proton g-factor	g_p = 5.585 694 713(46)
Neutron g-factor	g_N = –3.826 085 45(90)
Muon mass	m_μ = 1.883 531 30(11) x 10^{-28} kg
Inverse fine structure constant	α^{-1} = 137.035 999 679(94)
Planck constant	h = 6.626 068 96(33) x 10^{-34} J s
Planck constant over 2π	$\hbar$ = 1.054 571 628(53) x 10^{-34} J s
Bohr radius	a_0 = 0.529 177 208 59(36) x 10^{-10} m
Bohr magneton	μ_B = 927.400 915(23) x 10^{-26} J T^{-1}

Other values:

1 inch ≡ 0.0254 m (exact)

1 drop ≡ .05 ml (metric system, exact. Other definitions exist.)

1 eV/particle = 96.472 kJ/mole

kiloton ≡ 4.184 x 10^{12} J = 1 Teracalorie

bar ≡ 100,000 N/m^2

atm ≡ 101,325 N/m^2 = 1.013 25 bar

torr ≡ 1/760 atm ≈ 133.322 N/m^2

Contents

Working Introduction to Microcontrollers

This *brief* introduction is intended for students with little or no programming experience, and no knowledge of microprocessors/microcontrollers. It is intended to give you a *basic* working knowledge of microcontrollers so you can write a real program as quickly as possible. You will likely want to go on to more advanced texts for more information. We cover these topics:

1. Binary information
2. Memory
3. Microprocessors and microcontrollers
4. Assembly language

Binary Information

Almost all computers today store information in binary: as a series of digits that are either 0 or 1. A single 0 or 1 is called a **bit**., short for Binary digIT. Several bits are usually grouped together to form a **byte**, which is almost always 8 bits. For example

0	1	0	0	0	0	0	1

2^8 = 256 possible bit patterns in a byte

As a number: 65

As a character: the letter 'A'

As a computer instruction: ??

The meaning of the bit pattern in a byte depends on how it's used: it could be a number, or a printable character, or a machine instruction, or anything you want it to mean. As a number, there is a standard way to interpret the bits: as a base-2 number:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	1

As a number: $1 \times 2^0 + 1 \times 2^6 = 1 + 64 = 65$

As a number in this form, a byte can hold a value from 0 to 255. Larger numbers, and fractions, require more complicated data structures. Negative integers use 2's complement; see Funky Mathematical Physics Concepts for a description of 2's complement numbers (physics.ucsd.edu/~emichels).

Binary bits are tedious to write, because they're so long. If they represent numbers, we usually write them in decimal. If they represent other things, we might write them in **hexadecimal**, which is base-16. Hexadecimal is a short-hand for bit patterns: we can write any pattern of 4 bits as a single hexadecimal digit. The hexadecimal digits are 0-9 and A-F. Their bit patterns are:

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Hence, we can write the number 0100 0001 as 41h, where the 'h' denotes hexadecimal. The byte pattern 1010 1111 = AFh. Note that hexadecimal does *not* mean the bit pattern represents a number (though it might). "Hex" (as it's called) is just a shorthand for bit patterns. Some authors use lower case letters for the digits A - F, but it doesn't matter which case you use. A single hex digit, i.e. a group of 4 bits, is sometimes called a nybble ('cuz it's like a small byte. Get it?) Also, hex is sometimes written with a "0x" prefix, e.g. 0xABCD = ABCDh. Single digit numbers are the same in hex or decimal, e.g. 7h = 7.

If a byte represents a character, there is a standard representation of all the common characters, called ASCII (American Standard Code for Information Interchange), pronounced ass'-kee. You can look up the details, but examples are 'A' = 41h, 'B' = 42h, ... 'Z' = 5Ah

Memory

Computers store data in **memory**, which is some device for storing and retrieving bit patterns. We are concerned here only with **addressable memory**: a sequence of locations, each of which can store some number of bits. For the 18F4520 family, the addressable memory stores bytes, or 8-bits, in each location. Here's an example of an addressable memory holding data (it's contents). On the left (in black) is decimal addresses and binary contents. On the right (in blue) is the same data displayed as hex addresses and contents:

Address	Contents
0	0100 0001
1	0100 0010
2	0100 0011
:	
4095	0101 1010

↔

Address	Contents
0h	41h
1h	42h
2h	43h
:	:
FFFh	5Ah

FFFh = $15 \times 16^0 + 15 \times 16^1 + 15 \times 16^2 = 4095$

Each location is identified by a number, or **address**, starting from 0, and in this example, going to 4095. This memory holds 4096 bytes of information.

From now on, we always use hex instead of binary.

Strictly speaking, addressable memory should be called RAM (Random Access Memory), but the term "RAM" has been distorted to mean a specific kind of addressable memory. Today, RAM means memory that the computer can write to and read from very quickly. It is the fastest form of memory for both writes and reads.

[Sometimes, memory holds chunks of data larger than a byte. These are called **words** (though "mouthful" would better fit the analogy). A word might be 16-bits, 32-bits, or other sizes. The word-size depends on the computer type. We don't care about words right now.]

Microprocessors

Computers are machines which execute a stored sequence of instructions (a **program**) to read in data, store it, process it, and write it out. A **microprocessor** is the heart of a computer on a single-chip (integrated circuit). A **microcontroller** is a simple microprocessor, coupled with a bunch of handy add-ons, on a single chip. The add-ons are things like memories, Analog-to-Digital converters (ADC), Digital-to-Analog converters (DAC), pulse-width modulators (PWM), timers, counters, UARTs (don't worry about it), etc.

Microcontrollers include addressable memory to store the instructions they execute (the program), and memory to store the data on which the instructions operate. Let us invent a simple microcontroller, named "Mike", which illustrates the basic principles of a real microcontroller. In Mike's case, both program and data are stored in RAM, which is easily written and read.

Besides the addressable memory, all computers include a small number of special memories which hold only 1 byte, or a few bytes. Each such "tiny" memory is called a **register**.

The machine instructions are encoded as bit-patterns, i.e. as bytes. For Mike, each instruction is a single byte. Different bit patterns represent different instructions. Mike has a 16-byte RAM for program and data. Note that some memory locations contain instructions telling Mike what to do, and some locations contain data, on which Mike will operate. Mike also has 2 special purpose registers, which we will call the "program counter" or "PC", and "working register" or "WREG" (both defined shortly). We define 4 instructions, each with an identifying number:

(0) ‘stop’, (1) ‘copy-to-wreg’, (2) ‘copy-to-ram’, (3) ‘add-to-wreg’.

We now write a program to add the contents of memory locations 8 and 9, and store the result in location 10 (Ah in hex), then add the contents of locations 11 (Bh) and 12 (Ch), and store the result in location 13 (Dh). To start, our memory for the program and data might look like this (all numbers in hex):

	Address	**Contents**	**Meaning**	
PC = 0	0	18	copy-to-wreg	Instructions
(All values hex)	1	39	add-to-wreg	
	2	2A	copy-to-ram	
	3	1B	copy-to-wreg	
	4	3C	add-to-wreg	
	5	2D	copy-to-ram	
	6	00	stop	
	7	??	not used	
	8	11	data	Data
	9	22	data	
	A	??	don’t care	
	B	45	data	
	C	67	data	
	D	??	don’t care	
	E	??	not used	
	F	??	not used	

We explain the “contents” shortly.

Recall that the computer executes instructions in sequence, one after the other. The computer uses to the PC to keep track of which instruction it is executing. Since Mike has only 16 locations of RAM, and 4 bits can represent any number from 0 to 15, Mike’s PC is a 4-bit register, which holds the address of the next instruction to execute. When Mike completes execution of an instructions, it adds 1 to the PC, fetches the next instruction pointed to by the PC, and executes the new instruction.

When we reset Mike, it sets the PC to the number 0, and starts executing. Therefore, we start our program at address 0. The first nybble of each instruction is the instruction type, as given above: 0 = stop, etc. The 2nd nybble of each instruction tells Mike what memory location to use for the instruction (if any).

Most computers cannot directly add two numbers in memory. Instead, the computers define a register (e.g., a single-byte memory such as WREG) in which all operations take place. To add two numbers that are in memory, we must first copy one number from RAM to the WREG, then add the 2nd number to WREG, then copy WREG (now holding the sum) to RAM.

Let us now become one with Mike, and execute our program from the memory contents given above:

1. On reset, Mike sets the PC = 0
2. Mike executes the instruction at location 0: the first nybble is 1, which tells Mike to copy from RAM to WREG. The 2nd nybble is 8, which tells Mike to copy RAM location 8 to WREG. Now WREG = 11h, because RAM location 8 has the value 11h.
3. Mike adds 1 to PC, so PC = 1. Mike executes the instruction at location 1: the first nybble is 3, which tells Mike to add a value from RAM to WREG. The 2nd nybble is 9, so Mike adds the contents of RAM location 9 to WREG. Now WREG = 33h (which is 11h + 22h).
4. Mike increments PC to 2, and executes that instruction: the first nybble is 2, which means copy WREG to RAM. The 2nd nybble is Ah, so Mike copies WREG to RAM location Ah. Now RAM location Ah also contains 33h.
5. Mike increments PC to 3, and executes: the instruction is 1Bh, which means copy RAM location Bh to WREG. Now WREG = 45h.
6. Mike increments PC to 4: the instruction is 3Ch, which means add RAM location Ch to WREG. Now WREG = ACh (which is 45h + 67h).

7. Mike increments PC to 5: the instruction is 2Dh, which means copy WREG to RAM location Dh. Now RAM location Dh = ACh.
8. Mike increments PC to 6: the instruction is 00h, which means "stop". Mike stops.

In a real computer, a single instruction could be 1 byte, 2 bytes, 3-bytes, and sometimes even more. The computer knows how long an instruction is from the bit patterns which compose it.

Assembly Language

Programming Mike in binary (or even hex) is tedious, error prone, and hard to read. Mike has simple, well-defined instructions. Wouldn't it be nice if we could write our program's instructions in a human-readable form? An **assembler** is a program you use on your PC that converts human-readable instructions into machine-readable instructions that Mike (or some other computer) can execute. Every computer type (or family) must have its own assembler, because the assembler must know the details of the computer instructions, memory, registers, etc.

The first step in making our Mike program readable is to use short mnemonics (human-memory aids) for the instructions. Assembler instructions are often written in capitals. So our assembler instructions might be STOP, CPRW (copy RAM to WREG), CPWR (copy WREG to RAM), and ADD. Then our program could be written:

```
CPRW 8                  [‘CPRW’ tells Mike what to do, ‘8’ is the memory address to do it on.]
ADD  9                  [This ADD assembles into the instruction 39h, as above.]
CPWR Ah
CPRW Bh
ADD  Ch
CPWR Dh
STOP
```

This improves our instructions, but doesn't help define our data. The assembler provides a **directive** (also a mnemonic) to define data. The directive is DATA. Also, the assembler allows us to put comments on each line, after the "operands", or memory locations. So we could write:

```
CPRW 8      base price
ADD  9      add tax
CPWR Ah     total cost
CPRW Bh     mass of elevator
ADD  Ch     mass of person
CPWR Dh     total mass to lift
STOP
DATA 00     not used
DATA 11h    base price
DATA 22h    tax
DATA 00     total cost
DATA 45h    mass of elevator
DATA 67h    mass of person
END         end of program
```

We snuck in the END directive, which tells the assembler that the program is done. The END *directive* is very different from the STOP *instruction*, which causes Mike to halt when it executes. Notice that putting each DATA byte separately is tedious, so the assembler lets us combine them, by separating data values with a comma:

```
DATA 00,11h,22h,00          not used, base price, tax, total cost
DATA 45h,67h                mass of elevator, mass of person
```

We don't need to define location Dh, because it will be overwritten by the program when it executes.

This is already a big improvement, but having to write the RAM addresses as numbers on all our instructions is very bad: it's tedious, error prone, and makes changing your program very difficult. Of course, the assembler helps with this, too. The assembler allows you to give "names" to RAM locations (and other things). Then you refer to the RAM locations by name, rather than by number. The assembler converts the names to numbers for you when it "assembles" your source code into machine instructions. You can code a label at the beginning of most any assembly line.

The assembler assigns the *address* of instructions or data to the label beginning the line of code.

In addition, entire lines starting with ";" are comments. So we have:

```
;This program computes the total cost of a pencil, and the total mass of
; a graduate student in an elevator.
;Mike resets to 0, so this program must start there.
            CPRW   baseprice    base price
            ADD    tax          add tax
            CPWR   total_cost   total cost
            CPRW   m_elev       mass of elevator
            ADD    m_person     mass of person
            CPWR   m_total      total mass to lift
            STOP
            DATA   00           don't care
baseprice   DATA   11h          pencil base price
tax         DATA   22h          tax
total_cost  DATA   00           pencil total cost
m_elev      DATA   45h          mass of elevator
m_person    DATA   67h          mass of person
m_total     DATA   00           don't care
            END
```

The assembler assigns the value 8 to 'basepr', because 'basepr' is at address 8 in memory. Similarly, 'tax' = 9, 'total_cost' = Ah, 'm_elev' = Bh, 'm_person' = Ch, and 'm_total' = Dh. This program assembles into the same program that we originally wrote in hex, but is much easier to read, understand, and modify.

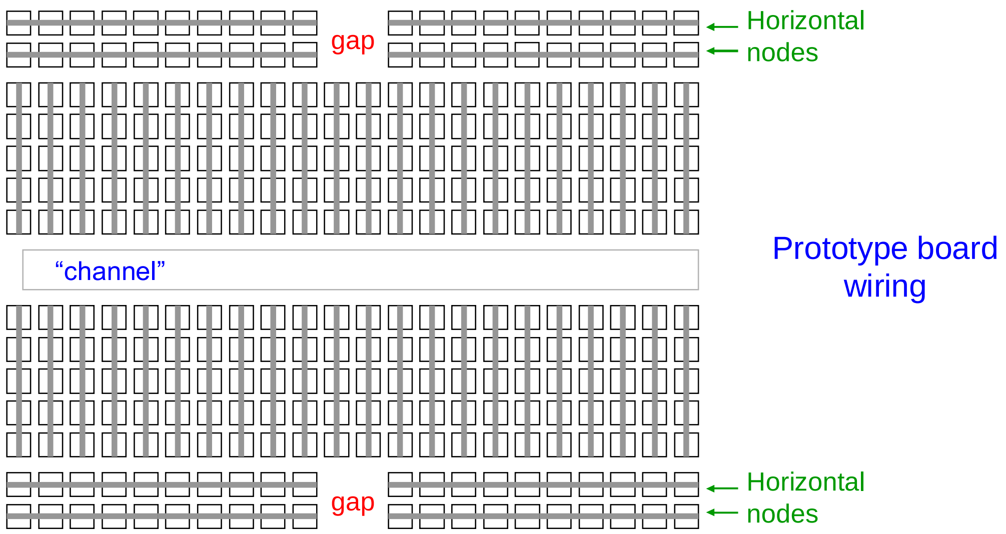
Notice that we indent our code so that all the instructions and directives line up, as do the comments.

You might wonder how the assembler can know the address of a label, such as 'baseprice' above, *before* the label appears in the code. It can't, and so many assemblers are "two-pass assemblers": they read the code twice. On the first pass, they determine the values of all the labels. On the 2nd pass, they generate the machine instructions.

TBS: 18F4520 information.

Prototype Board Construction

Prototype boards are commonly called “proto-boards” or “breadboards”.



The horizontal runs at the top and bottom are usually used for power and ground. The vertical runs, above and below the “channel”, are used for general interconnect. Note that for all but the smallest breadboards, there is a gap in each of the horizontal runs, in the middle of the board. Thus there are 8 nodes of horizontal connection. It is common to jumper across the gap, thus making the entire horizontal run electrically one node.

Glossary

address	a number which identifies a memory location
assembler	a program that converts human-readable instructions into machine-readable instructions that a computer can execute.
computer	a machine which execute a stored sequence of instructions (a **program**) to read in data, store it, process it, and write it out.
memory	a device for storing and retrieving bit patterns
microprocessor	the heart of a computer on a single-chip (integrated circuit).
microcontroller	a simple microprocessor, coupled with a bunch of handy add-ons, on a single chip. Microcontrollers include addressable memory to store the instructions they execute (the program), and memory to store the data on which the instructions operate.
PC	a register which holds the address of the next instruction to be executed
program	a sequence of instructions for a computer.
register	a special memory which holds only 1 byte, or a few bytes.