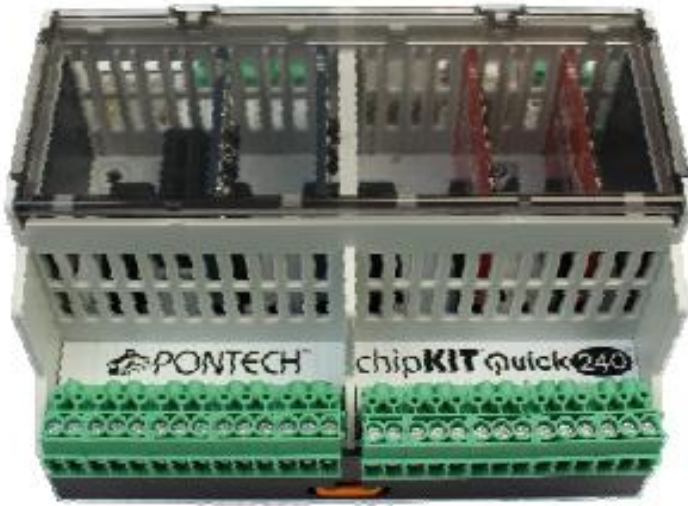


An Introduction to Embedded Systems

A cross platform approach using the Wiring programming paradigm with practical examples utilizing Arduino IDE with chipKIT and Arduino hardware.

By: Jacob Christ



Lecture 1 – Embedded Development Tools and Overview

Class Goals / Accessing Prior Knowledge

This material was (and is being) developed over several years for a second year Electronics Class at Mt. San Antonio College (affectionately known as Mt. SAC) in Walnut California. The class as listed in the course catalog at the college as such that there is no collegiate prerequisite. Being as such, any student should be able to walk into this class and be able to navigate through the material presented and increase their understanding on of the subject matter. There is a caveat though, nearly every student that comes through the class has at least a year of basic electronics under their belt. In the class if a student doesn't have the prerequisite electronics experience needed for specific understanding the instructor and fellow students can bring them up to speed quickly. Additionally, mastery that can be achieved by studying this material is greatly affected by an individual's prior knowledge before entering the class. This creates a rift between the highest achievers in the class and the lowest achievers. A clear demarcation between the two groups is that the students at the top of the rung have had prior experience writing computer software (in no specific language).

Some assumptions are made on my part based on my observation of the state of the common knowledge of the world today. Here is a list of things I suspect you probably know. This list is given so that if there is something you haven't seen or don't understand you can get yourself up to speed quickly so as to not feel overwhelmed by the class.

Prior Electronics Experience

Almost every student that enters this class has taken classes in DC (ELEC 50A at Mt.SAC) AC (ELEC 50B at Mt.SAC) and Digital Electronics (ELEC 56 at Mt.SAC).

Depending on the given year, about 25-50% of the students will have taken a class on electronic devices (ELEC 51 at Mt.SAC). The devices classes covers diodes, rectifiers, regulators, power supply design, transistor and op amp circuits.

A few will have taken microwave communications or industrial electronics classes.

Again, none of these classes are necessary, but knowing the subject matter from these classes frees your mind to think about new concepts rather than having to learn material that may have overlapped with these other classes in the given time we have in this class.

Prior Math Experience

Most (if not all) prior students have taken algebra but it drops off rapidly as we move from algebra to geometry to intermediate algebra to college algebra to trigonometry to pre-calculus and finally to calculus. Classes have had as few as one student that has had calculus and other times more than half the class has had calculus experience. This class uses some concepts from arithmetic and algebra. In lectures I will share some calculus concepts with the class but they should be refreshingly easy for

anyone with less than a calculus background, in fact the student would probably not even know its calculus if not pointed out.

Prior Computer

I suspect that everyone reading this material will have at least seen and used a computer sometime in their life. The skill levels are varied among students, but the following are necessary skills that if you are weak in you should seek immediate help.

Navigating the file system with a GUI tool (such as Explorer)

- Changing Directories

- Moving Files

- Copying Files

- Deleting Files

Internet

- Using a web browser

- Sending and Receiving e-mail with file attachments

- Downloading files from the internet

Running (Launching) Programs

Installing Programs

What are we in for?

The following is an incomplete list of projects that are in the same vein of what this course is all about. Let's take a look at some videos to get inspiration:

RC Servo Control: <http://www.youtube.com/watch?v=q2GlwfWwr3w>

Stepper Motor Control: <http://www.youtube.com/watch?v=09ckLqR05zA>

Stepper Motor Control: <http://www.youtube.com/watch?v=wypy1KdI5a8>

Super Chase: <http://www.youtube.com/watch?v=kiaJECGEtAM>

Light Cube: <http://www.youtube.com/watch?NR=1&v=GUCX41pokZY&feature=endscreen>

If the light cube intrigues you, then it might be worth noting here that by then end of the second lab you will understand the electronics to complete this project and the astute student will understand this by then end of lab one.

Function Generator: http://www.youtube.com/watch?v=gz_gVKWFN8E

Bike Computer: http://www.youtube.com/watch?v=O5YYsm_BqJQ

LED Clock: <http://www.youtube.com/watch?v=q4RwP0UK8gM&feature=endscreen&NR=1>

MakerBot: <http://www.youtube.com/watch?v=zkpPg84hxZg&list=PL16315F3CB6CF8637&index=2>

CNC Machine: <http://www.youtube.com/watch?v=SzFpiU6OSTQ>

Spider Robot: <http://www.youtube.com/watch?v=7L7oxoZEG-A>

Arc Reactor: http://www.youtube.com/watch?v=W_Jsu_E0go0

Music Visualizer: http://www.youtube.com/watch?v=__XwMbhV4k

Drum Machine: <http://www.youtube.com/watch?v=chEg6mAfaNA>

Sound Generation: <http://www.youtube.com/watch?v=nY1eQk3ezjM>

These videos are for inspiration. In this class we will learn some fundamentals required to build these types of projects but may not actually build anything like these.

Overview of an Embedded System

From wikipedia: http://en.wikipedia.org/wiki/Embedded_system

An **embedded system** is a [computer system](#) designed for specific control functions within a larger system, often with [real-time computing](#) constraints.^{[1][2]} It is *embedded* as part of a complete device often including hardware and mechanical parts. By contrast, a general-purpose computer, such as a [personal computer](#) (PC), is designed to be flexible and to meet a wide range of end-user needs. Embedded systems control many devices in common use today.^[3] Embedded systems contain processing cores that are either [microcontrollers](#) or [digital signal processors](#) (DSP).^[4]

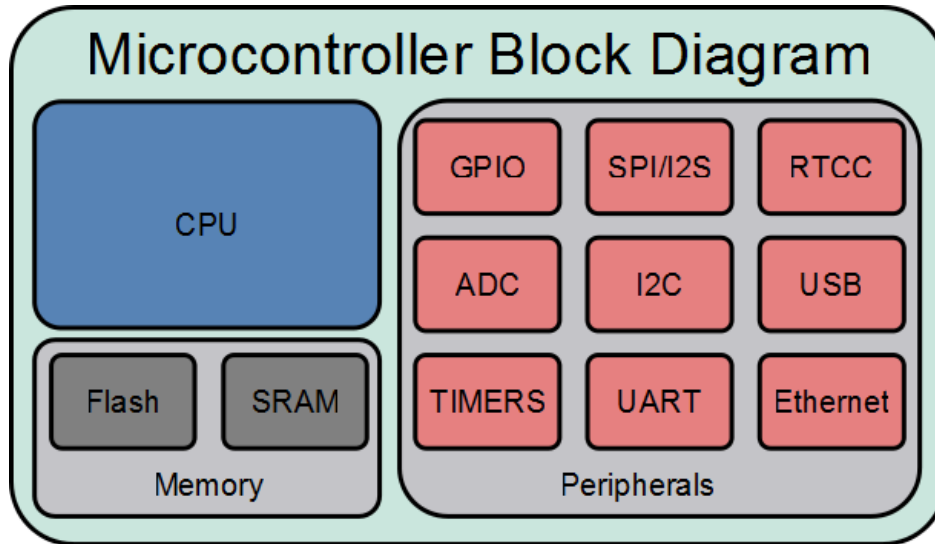
A processor is an important unit in the embedded system hardware. It is the heart of the embedded system.^[5]

The key characteristic, however, is being dedicated to handle a particular task. Since the embedded system is dedicated to specific tasks, design engineers can optimize it to reduce the size and cost of the product and increase the reliability and performance. Some embedded systems are mass-produced, benefiting from [economies of scale](#).

Physically, embedded systems range from portable devices such as [digital watches](#) and [MP3 players](#), to large stationary installations like [traffic lights](#), [factory controllers](#). Complexity varies from low, with a single [microcontroller](#) chip, to very high with multiple units, [peripherals](#) and networks mounted inside a large [chassis](#) or enclosure.

Introduction to Microcontrollers

A microcontroller is a monolithic device that contains a CPU (Central Processing Unit), nonvolatile memory (such as flash), volatile memory (such as SRAM) and a variety of peripherals devices.



They can be as small or smaller than 5 pin SOT-23 devices and as large as or larger than 144 pin BGA and TQFP parts.



The reason for choosing to use a microcontroller for a design would be to gain the advantages of a reprogrammable digital circuit in a form factor suited for a specific application when a general purpose computing device is not able to complete the task due to size or cost.

Abuse of terminology (CPU acronym)

CPU is an acronym for Central Processing Unit and is often used to refer to a computer system, such as "That beige box sitting next to my 24" flat screen monitor is my new CPU." The "beige box" being referred to in the aforementioned statement is a computer system and not a CPU, the CPU is the chip inside the computer system known specifically as the microprocessor. As technology advances and

desktop computer become rarer as laptop and tablets take over this abuse of terminology seems to be decreasing.

CPU Abstraction as a Microprocessor

Prior to the invention of the microprocessor in 1971 by Intel (the 4004) CPU's were circuits consisting of many chips to make up the function of a programmable information processing and manipulation device.

<http://www.intel.com/about/companyinfo/museum/exhibits/4004/index.htm>

Computer System Abstraction as a Microcontroller

A microcontroller is a monolithic device that contains a microprocessor and all the peripherals necessary to make a complete system on a chip. Microcontrollers make building a complete system a simple task by adding only a few additional components to bring a design to fruition.

Microprocessor

Information processing unit of the chip often referred to as the CPU. The CPU reads the stored program in program memory, interprets the instructions that are primarily used to direct the flow of information into and out of data memory and into and out of peripherals. Additionally, program flow control is handled by the microprocessor.

In desktop and laptop computers, the microprocessor is typically a standalone chip and all the additional features that make up a microcontroller are relegated to other chips that typically sit on the mother board. The ability to put different technologies onto a single chip is blurring what traditional computer architecture and an embedded architecture look like. Some computers on a chip have existed for quite some time that allow for incredible size reduction of computers allowing the creation of palm top computers.

Program Memory

Program memory is a non-volatile memory used to hold the application or operating system program for the embedded device. Non-volatile refers to the ability for the memory to retain its value without power applied to the chip.

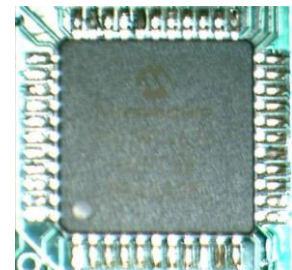
The program stored in the non-volatile memory of a microcontroller program memory is often referred to as firmware as opposed to software that would be loaded off of a hard drive on a PC.

Examples of types of non-volatile memory are:

Monolithic Device

mono = one, lithic = stone

The term monolithic device refers to the entire part being built out of a single silicone chip. The implications are such that the manufacturing process is simpler. Simplicity implies reliability (single failure point) and affordability (single part to purchase).



ROM

Read Only Memory, programmed at the factory and not erasable

PROM or OTP

Programmable Read Only Memory or One Time Programmable memory that cannot be erased once programmed

EPROM

Erasable Programmable Read Only Memory

EEPROM

Electronically (byte) Erasable Programmable Read Only Memory

Flash Memory

Electronically (bank) Erasable Programmable Read Only Memory

Data Memory

Data memory is volatile memory used to hold program data while the program is running. Volatile memory will not retain its values without power applied. Volatile memory is often called

RAM or RWM

RAM (Random Access Memory) or **RWM** (Read Write Memory).

The term RAM is quite popular but doesn't quite tell the whole story since ROM type memory is also random accessible. The better term is RWM since the memory can be altered on the fly while a program is running without special considerations that are necessary to erase a ROM chip.

There are at least two common types of RWM; Static and Dynamic. Static RWM is built out of transistors and is very fast and priced accordingly. Dynamic ram is built out of capacitors and is not as fast as static RWM but can be built much denser and at a lower cost. Also the nature of storing values as charges in a capacitor has the added complexity of requiring a dynamic ram controller to keep the correct values refreshed while the memory is not in use.

Microcontrollers' internal data memory is typical static RWM. Dynamic RWM is most commonly used as the main memory of a PC.

Side note, the cache of a PC microprocessor is made of static RWM and is on the microprocessor, but this was not always the case. Cache memory of older PC's was external from the microprocessor.

Other Non-Volatile Storage

In addition to program and data memory, some microcontrollers have additional non-volatile storage that can be used to store program data that survives power outages.

Peripherals

The peripherals of the microcontroller are what really differentiate a microcontroller from a microprocessor. The peripherals are used for the control and sensing of many things on a single chip. Listed below are some common peripherals that can be found on a microcontroller. Do not be discouraged if you do not recognize all, or for that matter any, of the peripherals for that is the purpose of this text. However the list is by no means complete.

Parallel Ports

Serial Ports

UART

SPI

I2C

USB

Timers / Counters / PWM

Analog Circuitry

ADC's

Comparators

What can a microcontroller do?

What are the limits of your imagination?

Advanced Understanding: The Church-Turing Thesis

In the 1936's Alan Turing and Alonzo Church independently postulated that not all numbers can be computed. Church using what he called lambda calculus and Turing using a machine (later called a Turing Machine by Church). Turing went further saying that any number that is computable can be computed by his machine. Further, any machine that can emulate a Turing Machine can compute any number that is computable (given sufficient memory). Microprocessors, and therefore microcontrollers, can emulate Turing Machines and therefore, based on the Church-Turing Thesis, can calculate any number that is computable. There are caveats, such as the limit of the memory connected to the computer limits what is computable. Also, the thesis speaks nothing about the speed of the computation which is dependent on many things (processor speed, architecture of the computers, number of processors and efficiency of algorithm used).

Although the thesis states that a Turing Machine can compute any number that is computable it is important to point out that not all numbers are computable. Incomputable numbers cannot be determined with a Turing Machine.

There are several things that you should take away from this. The ability to solve a problem (in computation) is not limited by any device that can emulate a Turing machine. The embedded computers we use in this class can solve any problem that any computer can solve.

The PIC32MX440F256H microcontroller

The PIC32MX440F256H is an 32-bit PIC microcontroller. The speed of the CPU is 80MHz which roughly translates into 80 million instructions per second (MIPS). It has 32K Kilobytes of RAM and 256 Kilobytes of flash memory. An internal EEPROM: 1024KB.

Programmable External Interrupts
Input Change Interrupts
Capture/Compare/PWM (CCP) modules
Master Synchronous Serial Port (MSSP) module ((SPI and I2C)
Enhanced Addressable USART module
10-bit Analog Digital Converter
Watchdog Timer
Brown-out Reset



(For details, please check the PIC32 datasheets from www.microchip.com)

Microprocessor / Microcontroller Terminology

Here are some terms and definitions that might help you when reading other material related to microcontroller technology.

CISC - A complex instruction set computer (CISC, pronounced like "sisk") is a microprocessor instruction set architecture (ISA) in which each instruction can execute several low-level operations, such as a load from memory, an arithmetic operation, and a memory store, all in a single instruction. The term was retroactively coined in contrast to reduced instruction set computer (RISC).

RISC - The acronym RISC (pronounced risk), for reduced instruction set computing, represents a CPU design strategy emphasizing the insight that simplified instructions which "do less" may still provide for higher performance if this simplicity can be utilized to make instructions execute very quickly. Many proposals for a "precise" definition have been attempted; however, the term is being slowly replaced by the more descriptive load-store architecture (see below). Well known RISC families include Alpha, ARC, ARM, AVR, MIPS, PA-RISC, PIC, Power Architecture (including PowerPC), SuperH, and SPARC.

Harvard Architecture – computer architecture with physically separate storage and signal pathways for instructions and data.

The Von Neumann Architecture - a processing unit and a single separate storage structure to hold both instructions and data.

Phase Lock Loop (PLL) - use as a Frequency Multiplier. For users who wish to use a lower frequency oscillator circuit or to clock the device up to its highest rated frequency from a crystal oscillator.

The Tool Chain


What is Arduino IDE?

The Arduino IDE is a software tool that is used to develop C and C++ code for embedded development boards such as Arduino and chipKIT boards we used in this class. IDE is an acronym for Integrated Development Environment. The Arduino IDE is a combination of technologies that allow editing of text “source code” files, compilation of “source code” into “machine code”, transferring the compiled machine code from our development machine to our target board and finally interacting with our target board through a serial terminal.

The technical process (as opposed to the creative process) of creating a program that runs on embedded hardware is as follows:

1. Enter source code in a text editor.
2. Compile source code into object code.
3. Link object code together into executable machine code.
4. Upload machine code to the target platform.
5. Test machine code in target application.

The Arduino IDE with the chipKIT-core installed utilizes gcc open source C++ cross compilers that have similar functionality that you would find in a C++ compiler for writing applications for a PC but utilizes libraries specifically written for embedded hardware. Programming is done utilizing the C++ language but automatically included libraries allow a beginner to ignore (at least in the beginning) the complexity associated with C++ so that they can dive in and get started creating fast.



The screenshot shows the Arduino IDE interface with the 'Blink' sketch open. The code is as follows:

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  Most Arduinos have an on-board LED you can control. On the Uno and
  Leonardo, it is attached to digital pin 13. If you're unsure what
  pin the on-board LED is connected to on your Arduino model, check
  the documentation at http://www.arduino.cc

  This example code is in the public domain.

  modified 8 May 2014
  by Scott Fitzgerald
  */

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);           // wait for a second
  digitalWrite(13, LOW); // turn the LED off by making the voltage LOW
  delay(1000);           // wait for a second
}
```

The status bar at the bottom indicates 'PONTech quick240 on COM3'.

Concepts in C Programming

Comments

Good programs will contain comments as necessary to make the code easier to interpret. It is important for other programmers to understand the logic to make modifications to your code, but also for yourself. You may have discovered in your past programming experience that if you go back to make changes to a program that you've written a while back. It may be hard to follow quickly some of the steps in the code.

A comment line can be included in C by bracketing all information that you don't want the compiler to translate with slashes and asterisks.

Comments / */*

```
/*
```

Anything between the slash-star star-slash characters is a comment and ignored by the compiler

```
*/
```

Comments //

```
// Anything after the slash-slash is a comment until
```

```
// the end of the line. Comments are ignored by the compiler
```

Brackets (or Braces)

Braces { } mark the beginning and end of a section of code. All code in-between the braces can be taken as at the same level of execution. Braces are often nested within "higher level" braces and these nested braces are used to group a section of code associated with a loop or a decision making code. The concept of braces and how they work will become more clear as the class progress and you see more code examples.

Put yourself in someone else shoes (Claude Shannon)

Imagine a time long ago before the advent of computers, digital logic or even transistors. Of course, this also means there are no electronic calculators. Relays, batteries switches and incandescent lights do exist. With just these components you set out to build a machine that can add two numbers together and display the result on a light. You have to use switches to represent your numbers inputs to the machine, relays to do your math and lights to display the results. Having never seen such a machine how would you use switches to represent numbers and how would you used relays to do math?

Week 2 – Number Systems, Output and Introduction to C

Links

Exponent Review

http://www.youtube.com/watch?v=joKpSP_-QFs

Number Systems

<http://www.youtube.com/watch?v=NlvZx2zsF1A>

Conversion Between Number Systems

<http://www.youtube.com/watch?v=fX61osaZa5w>

Class Goals (Lecture Name: Bits, Bytes and Data Types) (Functions and Loops)

This lecture should be about half review from digital, and half new

Counting and Number Systems

Review of Exponents

Let's say you're a mathematician and you spend your days multiplying numbers together, but not any number, you have this particular obsession of multiplying the same number to itself over and over such as $10 * 10 * 10 = 1,000$ or $10 * 10 * 10 * 10 = 10,000$.

After awhile of doing this you get tired of writing the same numbers over and over and because you're a good mathematician (which means your lazy and creative) you come up with a scheme so that you don't have to write so much. If you multiplying three tens together you decide to just write ten once then raise the number of times you want to multiply ten together in the upper right hand corner of the ten. You ponder this a bit you decide that this system will be really powerful and playing on the this you decide to say that $10 * 10 * 10$ becomes 10^3 (ten to the power of three). These are equivalent, and again were lazy so rather than writing equivalent we just write:

$$10 * 10 * 10 = 10^3$$

After awhile of playing around with this notation we decide we can generalize it in the following way:

$$b^n = b_1 * b_2 * b_3 \dots * b_n$$

And because we love math so much and we want to talk about our new notation with our math friends we come up with clever names and decide that in b^n that b will be called the base and n will be called the exponent.

So we continue playing with our exponents and bases and find a sticky situation comes up... What if our exponent is zero???

Well if our base is zero this is easy because we know zero times anything is zero:

$$0^3 = 0 * 0 * 0 = 0$$

But what about 10^0 ? Well this is zero 10's but should we say this is zero? What if we do the following?

$$10^3 * 10^0?$$

Isn't this just three tens multiplied by zero tens?

$$10 * 10 * 10 \text{ (and zero more tens?)}$$

If we decide that 10^0 is zero then $10 * 10 * 10 * 0$ is 0 not 100. It would probably be better to say 10^0 is 1 then $10 * 10 * 10 * 1$ is 100. And indeed, there is a rule that goes something like $b^n * b^m$ is $b^{(n+m)}$.

And we generalize that b^0 is one where b is any number... Then what about 0^0 ? Should this be one? Zero times anything is zero??? We'll leave this as an open question since it is beyond the scope of this class.

Exponents are essentially a short hand notation for multiplying the same number together over and over in succession and you should not be intimidated by them.

Counting in Decimal

Powers of 10

$$123 = 100 + 20 + 3 = 1 * 100 + 2 * 10 + 3 * 1 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0$$

Bits, Bytes and Nibbles

Review of a bit

Binary digit, has one of two possible values: 0 or 1

Review of a byte

1 byte is 8 bits written out left to right MSB (Most Significant Bit) to LSB (Least Significant Bit). Example

$$1001\ 0001 \text{ (base 2)} = 145 \text{ (base 10)}$$

Nibbles (no joke)

1 nibble is 4 bits

1 byte is 2 nibbles, usually referred to as the high nibble and the low nibble. When writing binary numbers, as above, a space is usually placed between nibbles to make the number easier to read by a human.

Counting in Binary

How many values (not how high can we count) with one bit?

Two, 0 or 1.

How many values (not high can we count) with two bits?

1: 0 0
2: 0 1
3: 1 0
4: 1 1

Two values for bit 1 and two values for bit 2. This can be calculated in the following manner:

This can be generalize as base^{digits}. The base is the base of the number system, in this case 2 and digits is the number of digits we have.

$$2^2 = 2 * 2 = 4$$

How many values (not high can we count) with eight bits?

$$2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 256$$

Hexadecimal (Hex for short)

Hexadecimal is the combination of the two words hex (meaning six) and decimal (meaning ten).

$$6 + 10 = 16$$

So in hex we have sixteen digits.

Count In Hex

In hex there are sixteen hexadecimal digits. We borrow the ten decimal digits we need six symbols to represent all sixteen hex values. We use the first six characters from the alphabet for this: A, B, C, D, E and F.

Recall when you count in decimal you start at zero and advance to nine using all ten digits before using another column, so a count sequence would look like this:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12...

The same is true in hexadecimal.

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12...

Hex digit / nibble equivalency

Recall a nibble is four bits and that four bits can represent sixteen unique values and since a single hex digit can be one of sixteen unique values then a hex digit is a nibble.

Summary of Counting

Decimal (base 10)	Hexadecimal (base 16)	Binary (base 2)	Octal (base 8)
0	0	00000	0
1	1	00001	1
2	2	00010	2
3	3	00011	3
4	4	00100	4
5	5	00101	5
6	6	00110	6
7	7	00111	7
8	8	01000	10
9	9	01001	11
10	A	01010	12
11	B	01011	13
12	C	01100	14
13	D	01101	15
14	E	01110	16
15	F	01111	17
16	10	10000	20

I/O PORTS

Our First Program (Blink)

Below is the typical blink example that ships with Arduino IDE. This program is pretty close to the simplest useful program that will run using the Wiring programming abstraction. The program when run will blink an LED attached to the development board at a rate of 0.5 Hz (one second on and one second off). If we analyze the code we can see it does one other very important thing, in the `setup()` it sets the pin attached to the LED to an output using the `pinMode()`. Most pins on a microcontroller have more than one function and almost every pin can be configured as digital input or digital output.

This program was originally written for the Arduino Uno and we make this assumption based on PIN 13 being used. On the Arduino Uno (and the chipKIT Uno32) PIN 13 is connected to an LED mounted on the development board.

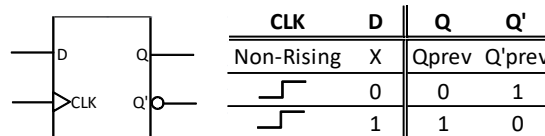
```
void setup() {
  // initialize the digital pin as an output.
  // An LED connected on most Arduino boards:
  pinMode(PIN_LED1, OUTPUT);
}
void loop() {
  digitalWrite(PIN_LED1, HIGH); // set the LED on
  delay(1000); // wait for a second
  digitalWrite(PIN_LED1, LOW); // set the LED off
  delay(1000); // wait for a second
}
```


Review from Digital

To understand the electronics behind the ability of the I/O pin on a microcontroller to be reprogrammable we first need to review some topics from digital electronic class. The 7400 series of TTL ICs contain AND gates, OR gates, NOT gates, etc. The inputs and output functionality of most of these device pins are fixed (as opposed to microcontrollers in which most of the pins can be programmed to be either digital input or digital output on the fly).

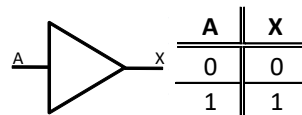
D Flip Flops (review from Digital)

The D flip flop is a very simple one bit memory circuit that remembers the bit value when upon a clock transition.



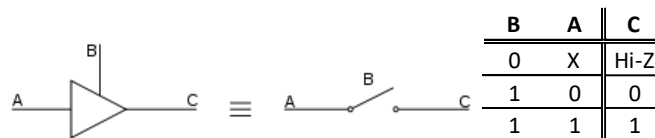
Buffers (Newish)

A buffer is simply a device that re-amplifies a signal and in the case of a digital buffer you get out the same logic level you put in. This circuit provides no enhanced ability to manipulate information, but gives us the ability to transmit a signal over a longer distance or to drive a larger number of inputs than would be possible with a single output.



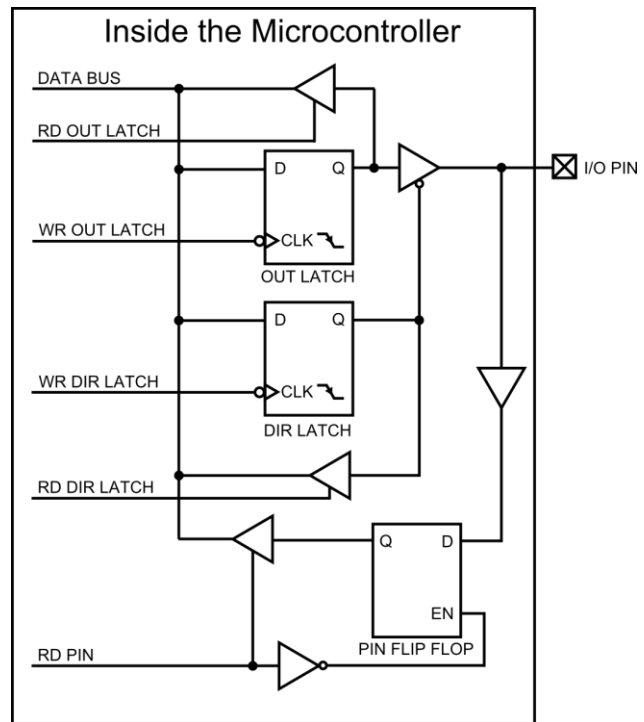
Tri-State Buffers (New)

Tri-state buffers have two inputs and a single output that can be in one of three states: High, Low or Hi-Z (high-Impedance). When the gate is in Hi-Z the gate is acting as if the output has been removed from the circuit. In reality it is still connected, but the drain on the rest of the circuit is so low that it can be neglected in simple cases.



I/O Pin Schematic of a Microcontroller

The input output functionality of the microcontroller is achieved by a clever arrangement of logic gates, flip flops and tri-state buffers. Shown below is a simplified diagram for a single bit of a microcontroller I/O port. The DATA BUS is a bi-directional signal buss where data travels to and from the microprocessor sub module of the microcontroller. The data bus connects to one of five locations within the I/O pin control circuit, two are inputs and the other three are outputs. The inputs are the D inputs to the OUT and DIR latches. The three outputs allow reading back the two latches and reading the raw digital value off of the I/O pin itself.



Data Direction Register (pinMode)

When the pinMode command is issued this causes a value to be written to the DIR latch that chooses if the I/O pin is going to be input or output. In the above diagram a logic 0 on the Q output of the DIR latch causes the port to become an output port, where a logic 1 on the Q output of the DIR latch disconnects the OUT latch from the I/O PIN allowing it to drive the input buffer that goes to the PIN FLIP FLOP. The state of the OUT latch can be read by using the getPinMode() command.

I/O Register (digitalWrite / digitalRead)

As seen in the blink sketch example from above, once the pinMode() for a specific pin has been set the output state of the pin can be set using the digitalWrite(). Equivalently, if we the pinMod() set the I/O pin to be an input then the digitalRead() can be used to read the logic state of the pin.

Questions

Is the value of a port volatile or non-volatile?

Concepts in C Programming

The nature of this class

In this class we are writing programs. An example of a programs are a word processor or spreadsheet or a web browser. Programs are instructions to make a computer do useful things. Useful has a very broad definition. A web browser lets you browse web pages on the internet and dose many other things. In the first few labs of this class we have written programs to flash LED's. Flashing an LED is a useful task, it may not seem very useful but have you ever seen an LED flash itself?

In this class we will be using the C/C++ programming language to write our useful programs.

Some History

Developed by K & R at AT&T Bell Labs for the purpose of developing operating systems. Designed to be a low level language (close to assembly) to harness the power of the machine but simplify the job of programming. (needs source)

The syntax used in C/C++ as well as other comes from this lineage of programming languages.

Alog60 BCPL -> B -> C -> C++, Objective C, Java

C Main Function

C is a functional language and the first function that gets called is main(). As used with the Wiring / Arduino / chipKIT paradigm the main() function is hidden from the developer and instead replaced with a setup() and a loop(). The setup() is called once at program start and the loop() function is called over and over as long as the device is powered.

Keywords or Reserved Words

Definition: List of words reserved by the language (C in our case) that have a predefined meaning.

auto d	entry	return	void d
break	enum d	short d	volatile d
case	extern d	signed d	while
char d	float d	sizeof	
const d	for	static d	
continue	goto	struct	
default	if	switch	
do	int d	typedef d	
double d	long d	union d	
else	register d	unsigned d	

C Reserved Words

Variables and Data Types

Variables symbols that represent the memory of the computer. An example of a variable definition is:

```
int a; // define a variable called a that is of type int
```

You can think of a variable as a named box that can hold a value and the above line of code does two things. One it names the box "a" and two it tells us how big we want the box to be, in this case the will hold an int.

```
int a  int a;
```

To continue with this analogy we need it would be nice if we can store something in our box. The C operator that allows us to store a value in a variable is the single equal symbol "=". The single equal symbol is called assignment.

```
a = 10; // Store the value of 10 in the variable a
```

```
int a 10 a = 10;
```

When a variable is defined a constraint is placed upon the variable as to the type of value that can be represented by the variable. In the above example the type is 'int' which means a signed integer value. By signed we mean that is can be either a value that is positive, negative or zero. Besides being signed an integer value could also be unsigned meaning that is can only represent zero and positive values. For review, an integer, is a whole number. That is to say is can be 1 or 2, but now 1.5 or 1 and 1/2. Another type of numerical value is a float which can represent fractional values such as 1.5 or 3.14159.

The data type is important does a couple of things for us. It tells the compiler how much memory to reserve for the variable being defined. It also allows the compiler to do sanity check such that we do not try to assign a floating point value to an integer value or something much worse.

A question that you should be asking right now is why are there so many different data types to represents a numerical value? The answer is probably most likely originating in the fact that early computer systems had very little memory so to represent a small number such as 15 you would need less memory than would be needed to represent a large number such as 128.25651210242048. Indeed, when we are developing with small microcontrollers memory too again is an issue we may be working with kilobytes of RAM compared to gigabytes in a modern computer. For this reason, it is important to choose a data type that will hold the largest and smallest number you want to represent but not one larger to consume precious memory.

Summary

C Language	stdint.h	Bits	Min	Max
bit		1	0	1
signed char	int8_t	8	-128 (-2^7)	127 (2^7-1)
char	uint8_t	8	0	255 (2^8-1)
int	int16_t	16	-32768 (-2^{15})	32767 (2^{15})
unsigned int	uint16_t	16	0	65535 ($2^{16}-1$)
long	int32_t	32	2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31}-1$)
unsigned long	uint32_t	32	0	4,294,967,295 ($2^{32}-1$)
long long	int64_t	64	9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 ($2^{63}-1$)
unsigned long long	uint64_t	64	0	18,446,744,073,709,551,615 ($2^{64}-1$)
C Language		Bits	Value Range	Precision
float	N/A	32	1.2E-38 to 3.4E+38	6 decimal places
double	N/A	64	2.3E-308 to 1.7E+308	15 decimal places
long double	N/A	80	3.4E-4932 to 1.1E+4932	19 decimal places

How to Represent (Some) literals in C

Bits

0 or 1 for single bit

0b00001111 as a byte

Decimal Numbers

0 to 255 for an unsigned byte

Hexadecimal Numbers

0x00, 0xa5 (or 0xA5) or 0xFF

Octal Numbers (noted only as a warning)

0777 is not 777

Assignment

a = b + 1;

Value to the left of = symbol is called lvalue and must be a variable. Values to the right of the = symbol can be variable, constants or literals.

Question: In this statement the microcontroller is reading from which values and writing which values?

#include

Includes a library in the C program

#include <system includes>

#include "local includes"

Assignments

Homework (Number Systems), Complete Lab 2

Week 3 – Input, Preprocessor, Loops, Decisions and Operators

Links

Number Systems Worksheet Solved

<http://www.youtube.com/watch?v=-y-EyWOXOJA>

Array's in C

<http://www.youtube.com/watch?v=wvrZpQWPvNg>

Bitwise Operators in C

<http://www.youtube.com/watch?v=5vJZ0-08FMY>

loops

We have seen in our sketches the setup and loop functions. Where the setup() will run one time on power up (or reset) and the loop() runs over and over in a loop after the setup() has run.

The loop() is very useful in that we do not typically want our program to just end after one pass of running the code, we want it to run over and over. In fact, looping is so powerful an idea that there are specific C language constructs for looping beyond this loop().

```
void setup() {  
    // run once:  
}  
  
void loop() {  
    // run repeatedly:  
}
```

The loop consists of a loop type, a code-block of one or more statements (which is to run over and over in a loop) and a condition that when met that allows the loop to exit. The condition for exit can be checked either at the top of the loop (before the code-block has a chance to run) or at the bottom of the loop (which allows the code-block to always run at least one time).

while(condition is true) { code-block }

The while loop does a conditional check at the top of the loop. If the condition is true then the program will be allowed to enter the code-block. After each time the code-block is run the program jumps back up to the top of the loop and checks the condition again. If the condition is still true the loop will run again. The condition can be chosen such that the loop will never exit, for example (1==1) is always true, so is (1) or (true). This can be useful, in fact this is how the function that calls loop() works. It can also be a source of errors in the program if our condition is not set up correctly.

```
uint8_t i;  
  
i = 0;  
while( i < 10 ) {  
    // do something  
    i++; // equivalency: i = i + 1;  
}
```


for(initial assignment; condition is true; incremental assignment) { code-block }

In addition to conditional checking the for loop has parameters for an initial assignment of an iterator and an incremental assignment. The initial assignment is run one before anything else. Then the condition is checked at the top of the loop just like a while. If the condition is true then the program will be allowed to enter the code-block. After each time the code-block is run the incremental assignment will be run then the program jumps back up to the top of the loop and checks the condition again. If the condition is still true the loop will run again.

```
uint8_t i;

for( i = 0; i < 10; i++ ) {
    // do something
}
```

for / while equivalency

The above two while and for examples are functionally equivalent. The C language could have gone without one of the two, but they are both available so both should be understood.

do { code-block } while(condition is true);

The do-while loop does a conditional check at the bottom of the loop. The code-block will run at least one time. After the code-block has run if the condition is true then the program will be allowed jump back to the top of the loop and run again.

```
uint8_t i;

i = 0;
do {
    // do something
    i++; // equivalency: i = i + 1;
} while( i < 10 );
```

Demonstration / Group Exercise

The following programs are done as a group exercise in class. They help demonstrate the how to use the Serial class and using variables to control looping in your programs.

Program 1 Serial Demonstration

This program is used to demonstrate that the messages sent over the serial port are lost until a serial terminal is opened. Once this program has finished uploading and is running on your board, as quickly as possible open the serial terminal. Notice that you will never see the message "Just opened the serial port." This is because the message is sent so fast after the Fubarino serial port has been opened that it's not possible to see the message. The five second delay after this message gives you just enough time to open the serial terminal and see the next message "5 seconds went by."

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  Serial.println("Just opened the serial port.");
  delay(5000);
  Serial.println("5 seconds went by.");
  delay(1000);
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.println("looping.");
  delay(5000);
}
```

Count from 0 to 10

This next program adds a while loop to the above program in the setup() to demonstrate how to automate counting from 0 to 10. The loop() has been cleared of any code. This results in the while loop running until completion then the program essentially halts. The new components of the program are the creation of a signed 8-bit variable named count:

```
int8_t count;
```

Initialization of the count variable to the value of 0.

```
count = 0;
```

The while loop that check to see if the variable count is less than or equal to the value 10.

```
while(count <= 10)
{
}
```

And the code inside the while loop (including the itterator). The itterator is the mechanism by which the loop keeps track of how many times it has executed as well as the means by which detection of the condition by which the loop will exit.

```
count = count + 1;
```

Here is the code:

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  delay(3000);
  int8_t count;
  count = 0;
  while(count <= 10)
  {
    Serial.print("The count is: ");
    Serial.println(count, DEC);
    delay(1000);
    count = count + 1;
  }
}

void loop() {
}
```

Write a program that out puts the numbers 0 to 20 then stops.

Each example from now on slightly modifies the above program. The changes to alter the above program are highlighted and made bold to help the reader identify the changes necessary to achieve the goals of the example. If you have not tried this code, you should run this code and make the changes for each successive version for yourself.

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  delay(3000);
  int8_t count;
  count = 0;
  while(count <= 20)
  {
    Serial.print("The count is: ");
    Serial.println(count, DEC);
    delay(1000);
    count = count + 1;
  }
}

void loop() {
}
```

Write a program that out puts the numbers 0 to 20 and counts by 2's then stops.

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  delay(3000);
  int8_t count;
  count = 0;
  while(count <= 20)
  {
    Serial.print("The count is: ");
    Serial.println(count, DEC);
    delay(1000);
    count = count + 2;
  }
}

void loop() {
}
```

Write a program that outputs the numbers 0 to 300 and counts by 10's then stops.

// First Try

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  delay(3000);
  int8_t count;
  count = 0;
  while(count <= 300)
  {
    Serial.print("The count is: ");
    Serial.println(count, DEC);
    delay(1000);
    count = count + 10;
  }
}

void loop() {
}
```

Write a program that outputs the numbers 0 to 300 and counts by 10's then stops.

// Second try

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  delay(3000);
  uint8_t count;
  count = 0;
  while(count <= 300)
  {
    Serial.print("The count is: ");
    Serial.println(count, DEC);
    delay(1000);
    count = count + 10;
  }
}

void loop() {
}
```

Write a program that outputs the numbers 0 to 300 and counts by 10's then stops.

```
// Third try
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  delay(3000);
  int16_t count;
  count = 0;
  while(count <= 300)
  {
    Serial.print("The count is: ");
    Serial.println(count, DEC);
    delay(1000);
    count = count + 10;
  }
}

void loop() {
}
```

Write a program that out puts the numbers 30 to 0 by -10.

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  delay(3000);
  uint8_t count;
  count = 30;
  while(count >= 0)
  {
    Serial.print("The count is: ");
    Serial.println(count, DEC);
    delay(1000);
    count = count - 1;
  }
}

void loop() {
}
```

*Note: If your variable is unsigned (uint8_t) it will never stop counting.

Review

Variables and Memory

The memory of the microcontroller (RAM) can be use in our programs to hold values that can be used to make decisions or to store data. How the memory is allocated for use is described by datatypes. The datatypes we have seen so far are bit, char (unsigned and signed) and long. The lab also mentioned arrays.

Class Goals

Preprocessor, headers, array's, sizeof operator, inputs, buttons, loops, if-else, logical and bitwise operators

Concepts in C Programming

Preprocessor Commands

Preprocessor commands are instructions in the a .c or .h file that a processed prior to code compilation, hence the name preprocessor. Preprocessor commands are easily identified since they start with a '#' (pound or sharp) character and do not require a ';' (semicolon) at the end of the command.

#include (header files)

The #include command causes another (code) file to be merged into the current file at the point where the statements has been added to the file. This allow for the reuse of libraries (of code). When programming the use of libraries is a powerful feature that allows the creation of complex program very quickly.

There are two types of libraries that can be included, system or compiler libraries (provided by the compiler creators) or user libraries.

System libraries are included by the use of the < and > delimiters for the file-path. An example of a system library we have already seen is the system.h file.

User libraries are included by the use of the " and " delimiters for the file-path.

#define

The #define command is used to create a reusable command that is often an abbreviated form of a more complex instruction. The #define is often called a macro command. For example:

```
#define uint8_t unsigned char
```

Allows you to declare a variable as

```
uint8_t var;  
rather than
```

```
unsigned char var;
```

The former being shorter and quicker to type than the latter.

Arrays

As we saw to create a single byte unsigned variable we can use the `uint8_t` macro to declare the x variable as such:

```
uint8_t x;
```

This reserves one byte of memory that can be used to store an unsigned 8-bit value. Recall that an unsigned 8-bit value is in the range of 0 to 255 (2^8-1 , where $n = 8$).

Likewise if we want to allocate 5 bytes of memory we can use the following declarations:

```
uint8_t x, xx, xxx, xxxx, xxxxx;
```

or

```
uint8_t x[5];
```

The first declaration creates five variables that can store five different values. The second declaration is called an array and does the same as the first, with the difference that they can be addressed (or indexed) using a second variable to select the position in the array.

In the statement: `uint8_t x[5]`, five memory locations are in the range of zero to four: `x[0]`, `x[1]`, `x[2]`, `x[3]`, `x[4]`. An individual value in an array is called an element. In the C programming language array elements always start at an index of zero.

The element `x[5]` is accessible, but is not valid since it is outside the range 0-4.

In addition to allocating the memory when a declaration is made, values can be assigned to them. The example below creates a variable array containing five values that have been assigned to each position in the array:

```
uint8_t x[5] = { 55, 75, 90, 40, 20};
```

This is equivalent to the following:

```
uint8_t x[5];  
x[0] = 55;  
x[1] = 75;  
x[2] = 90;  
x[3] = 40;  
x[4] = 20;
```

When you define a new array with initialized values, if you leave the array size blank then the compiler will calculate the proper size to make the array. The following example is the same as above but the size of the array has been left blank.

```
uint8_t x[] = { 55, 75, 90, 40, 20};
```

This is useful because you can add data without counting the number of elements.

sizeof()

The `sizeof` operator returns the number of bytes that a variable in the C language takes up in memory. For an array it will return the number of elements in an array.

```
uint8_t count_x;  
uint8_t count_y;  
uint8_t x[] = { 55, 75, 90, 40, 20};  
uint32_t y[] = { 55, 75, 90, 40, 20};  
count_x = sizeof(x); // sizeof(x) returns 5 since a uint8_t takes 1 byte  
count_y = sizeof(y); // sizeof(y) returns 20 since a uint32_t takes 4 bytes
```

The above example assigns the value of 5 to `count_x`. There are five one byte elements in the positions 0 to 4. The value of 20 is assigned to `count_y` since a `uint32_t` is 4 bytes long and there are 5 elements ($4 * 5 = 20$)

if, if-else and if-else if statements (decision statements)

“if, if-else and if-else if statements” provide the ability to alter program flow from its normal top to bottom execution based on a “condition” and take the following form:

```
if
  if(condition is true)
  {
    // run this code if condition is true
  }
```

if-else statements

```
if(condition is true)
{
  // run this code if condition is true
}
else
{
  // run this code if condition is false
}
```

iff-else if-else

```
if(condition 1 is true)
{
  // run this code if condition 1 is true
}
else if(condition 2 is true)
{
  // run this code if condition 2 is true
}
else if(condition N is true)
{
  // run this code if condition N is true
}
else
{
  // run this code if no condition is true
}
```

Comparisons Operators

Name	C Operator
Equal to	==
Not equal to	!=
Less than	<
Greater than	>
Less than or equal to	<=

Greater than or equal to	>=
--------------------------	----

“Conditions” in C are statements that evaluate a truth value of either TRUE or FALSE and in the form of A OP B, where A and B are values to be compared and OP is an operator from the chart above.

Condition Truth Values in C

Truth values in C are integers and therefore are actually represented in the computer as a numeric value. A value that is FALSE is the integer value zero and a value that is TRUE is not zero (anything other than zero). Most of the time, TRUE values are the integer one, but any non-zero integer is TRUE when being evaluated as a condition.

This is why the following while statement never exits:

```
while(1)
{
    // code that is run forever...
}
```

The while loop executes as long as the condition is TRUE, since 1 is not zero and any non-zero value in C evaluates to TRUE, this loop never exits.

Operators (Bitwise and Logical)

Operation	Logical	Bitwise
NOT	!	~
AND	&&	&
OR		
XOR	NONE	^

Bitwise

Bitwise operators compare individual bits of a value and result in a new value of the individual compared bits. This operation can be thought of as turning a truth table on its side.

```
// 0x6A    0110 1010      0110 1010      0110 1010
// 0x0F | 0000 1111      & 0000 1111      ^ 0000 1111      ~ 0000 1111
// -----
//          0110 1111      0000 1010      0110 0101      1111 0000
```

```
uint8_t a = 0x6a;
uint8_t b = 0x0f;
uint8_t c;

c = a | b; // c <= 0x6f
c = a & b; // c <= 0x0a
c = a ^ b; // c <= 0x65
c = ~b;    // c <= 0xf0
```

Logical

Logical operators evaluate two truth values and result in a single truth value, for example “true || false” will evaluate to “true”. In the example below remember that any non-zero value is considered true so the following:

```
// 0x6A    0110 1010      0110 1010
// 0x0F || 0000 1111      && 0000 1111      ! 0000 1111
// -----
//          1              1              0
```

Can be thought of as:

```
// 0x6A          1          1
// 0x0F ||      1      &&      1      !          1
// -----
//          1              1              0
```

```
uint8_t a = 0x6a; // non-zero, therefore true
uint8_t b = 0x0f; // non-zero, therefore true
uint8_t c;

c = a || b; // c <= 0x01 (true)
c = a && b; // c <= 0x01 (true)
c = !b;    // c <= 0x00 (false)
```

Shift operators

>> Shift bits to the right, << Shift bits to the left
 Shift bit n times.

```
// 0x6A      0110 1010      0110 1010      0110 1010      0110 1010
//          <<          1      <<          4          >>          1          >>          4
//          -----          -----          -----          -----
//          1101 0100      1010 0000      0011 0101      0000 0110
```

```
uint8_t a = 0x6a;
uint8_t c;

c = a << 1;    // c == 0xd4
c = a << 4;    // c == 0xa0
c = a >> 1;    // c == 0x35
c = a >> 4;    // c == 0x03
```

a = a << 1; is equivalent to a <<= 1;

a = a >> 1; is equivalent to a >>= 1;

Shifting left is equivalent to multiplying by two and shifting right is equivalent to dividing by two.

Logical Values in C

In the C programming language all values are numbers, these numbers if evaluated as a logical expression are either true or false. A value of 0 is false and a value of not zero is true. So...

0 is false

1 is true, but so is 2,3,4 or 5...

!0 is true

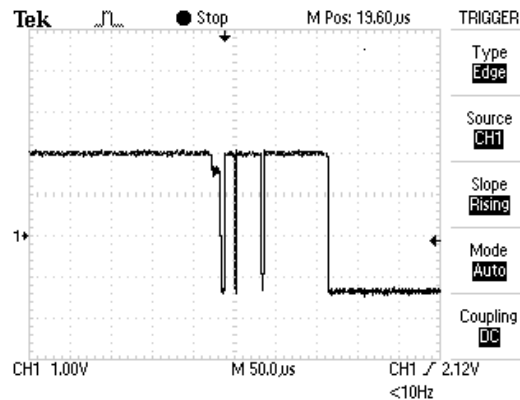
!1 is false, but so is !2, !3, !4 and !5... since 2,3,4 and 5 are true.

The reason why the statement while (1) never exits the loop is because 1 evaluates to true.

De-bounced Switch

If we create a logic switch out of a SPST switch and a pull up resistor. This crude circuit works fine for some tasks but when we need to count button presses it has a weakness. The physics of a button is such that as the button is pushed and the two conductive surfaces come into contact with each other they tend to bounce at first. This could be caused by several different phenomena such as the spring constants in the materials causing an oscillation or dirt on the contact surfaces causing momentary make then break affects at the beginning of the push stroke before a large enough contact patch is established for continuous current flow. In either case the result is the same, eradicate voltage swings

seen at the uC pin at the moment of contact. This can be seen by probing the uC pin with a digital capture oscilloscope and pushing the button.



If our microcontroller runs at 80MHz and can process an instruction in one cycle then this means that the execution time of a single instruction is 12.5ns. If we are trying to count button presses then a single button press, as shown in the image above, could be counted several times. This is usually not desirable and it completely unpredictable. This can be corrected for either in software by waiting several microseconds between looking for presses, or more commonly by adding a capacitor to ground (when a pull-up is used) or to Vcc (when a pull-down is used).

Using a de-bounced switch can be very important especially if we are using a button to trigger an interrupt (to be discussed later) since not doing so could trigger multiple interrupts for a single button press when we only desired one.

Week 4 – Functions, Program Flow and Interrupts

Review

Bitwise operations as masks...

Example 1 Bitwise OR as mask (|)

```
// 0x6A   0110 1010  <- Value  
// 0x0F   | 0000 1111  <- Mask  
// -----  
// 0x6F   0110 1111
```

High nibble passes through, low nibble masked to value one.

Example 2 Bitwise AND as mask (&)

```
// 0x6A   0110 1010  <- Value  
// 0x0F   & 0000 1111  <- Mask  
// -----  
// 0x0A   0000 1010
```

High nibble masked to value 0, low nibble passes through.

Class Goals

Functions (Subroutines or Subprograms)

The Call Stack

Functions (Subroutines or Subprograms)

We have already seen in the programs we have created function definitions called setup and loop which are used as entry points (where our program starts running) into our program.

```
void setup() {  
    pinMode(13, OUTPUT); // Function we call  
}  
  
void loop() {  
    digitalWrite(13, !digitalRead(13)); // Functions we call  
    delay(1000);  
}
```

Functions
we define

We have also use some functions defined in libraries included with the **Wiring API** called pinMode(), digitalWrite(), digitalRead(), delay() that provide functionality that we can leverage.

Finally, we have worked with functions definitions that are neither part of the Wiring API nor are they required. Examples include nibbleToPins().

The C programming language is a functional programming language. That is to say that the programs we write will be made up of lots of functions. All the functions we call will be called from one of our two entry points in to our program: setup() or loop().

In other programming languages, what we call a function in C/C++ is often referred to as a subroutine or a subprogram.

If we think of the term function in use outside of the C/C++ programming language, then you might think of a "The function of a device." Which is saying how the device works or what it is used for. You might also think of a math function such as $y = mx + b$. A function in C/C++ is much like a math function in that we can set some parameters and it can perform some operations on those parameters and then return a value that can be assigned to a variable. But it is also different in that within the function we can perform algorithms that manipulate data in a more complex form than simple math equations.

The C/C++ Entry Point

As we have seen, in the Wiring API there are two entry points into our program called the setup and loop. In the C/C++ programming languages there is a single entry point called main().

Every C/C++ program has a main function. When writing C/C++ code for the Wiring API (such as we do in chipKIT or Arduino) the main() is hidden from us. If we look at the main() we see that it is quite simple and just calls the setup() and loop() for us. The main function for chipKIT and Arduino looks something like this (depending on the version of the IDE you are running):

```
int main(void)
{
    init();
    setup();

    while (1)
    {
        _scheduleTask();
        loop();
        if (serialEventRun) serialEventRun();
    }
    return 0;
}
```

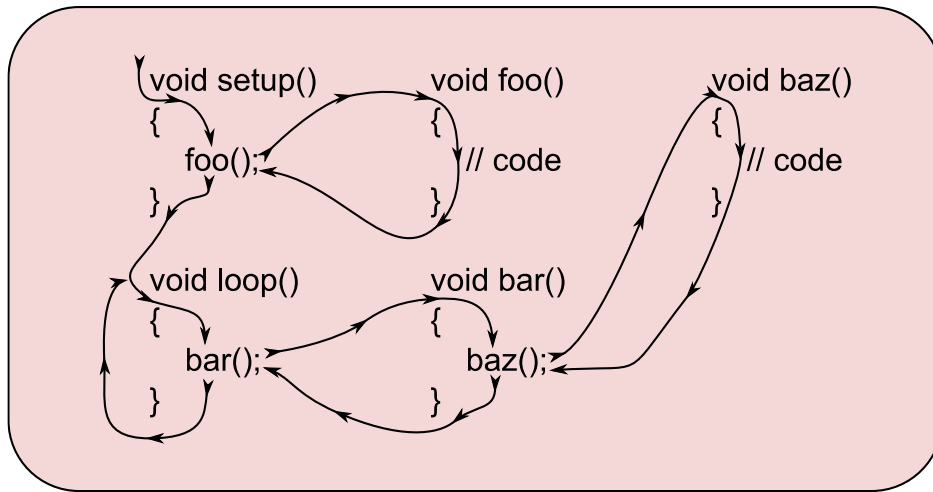
As you can see the main() calls four additional functions: init(), setup(), _scheduleTask() and loop(). The setup() and loop() functions are where the functions in our programs are called from. The init() and _scheduleTask() functions are called to manage internal housekeeping for the specific device we are using.

Arduino IDE (C Language) Program Flow

A function that uses another function is said to “call” the function. Each function that we call from main in turn may call additional functions itself. Below is a diagram showing the flow of a program that calls several functions as well as functions that call functions.

Metasyntactic Variables

To understand where the function names foo, bar and baz came from. Search for the term "metasyntactic variable" using the internet or other preferred reference source.



Functional Flow of a MPIDE Program

Function Prototypes

When we create a function in C, it is said that it must be declared prior to use. This literally means that code for the function must be located above (before) the function that calls it. So a program that calls a function `foo` must look like this:

```
void foo(void)
{
    // Some code
}
void main(void)
{
    foo();
}
```

But cannot look like this since `main()` calls `foo()` prior to declaration:

```
void main(void)
{
    foo();
}
void foo(void)
{
    // Some code
}
```

It is possible to have the `foo` function exist after the `main` function by use of a function prototype. A function prototype is a statement that tells the compiler that a function will exist and it has the following form, but the definition is yet to come. This will allow functions to call it without it being defined. Here is our second example again, corrected with a function prototype that is declared prior to `main()`.

```
void foo(void);

void main(void)
{
    foo();
}
void foo(void)
{
    // Some code
}
```

Notice that the function prototype looks the same as the first line of the function definition except that it is terminated with a semicolon.

ADVANCED UNDERSTANDING

The reason why a function must be declared prior to use has to do with the efficiency in which a C compiler converts a C program into assembly (or machine code). C Compilers are one pass compilers. This means that program is converted from C to assembly by reading each character in our source file only once. So when a function is called, if it was not declared previously the compiler does not know where to find it.

CONFUSION POINT

The words `declare` and `define` sound very similar to a non-programmer. But in programming they have very distinct meanings. To `declare` a function is to say that it exists, but not what it is. To `define` it means to say that it exists and this is what it is.

Arduino IDE does not require the use of function prototypes and is able to do this because the IDE scans your code and creates them for you prior to sending your code to the compiler. This feature was added to the IDE to simplify creating programs for beginners but comes at a cost (nothing is for free). As mentioned above, the purpose of function prototypes is let the compiler know that a function exists and will be seen later so that it can handle calls to functions prior to definition. Function prototypes are required by the compiler so Arduino IDE must create them for us. This means that the program must be scanned and manipulated prior to compilation this scanning takes time and results in the files being sent to the compiler not matching exactly what we see in our editor. The implication is we have to wait longer for our program to compile and when we get error messages from the compiler it will indicate that the error is on a different line of our code that it actually is.

Passing Parameter(s)

In addition to being able to call a function, we can all pass parameters (or arguments) to them, this useful to allow our functions to have greater capabilities. The number of parameters we can pass to a function is limited only to the amount of memory we have on our computer. The `setup()` and `loop()` booth take zero parameters, All other functions we have seen so far take one or more parameters. A function that requires zero parameters is declared by using the `void` keyword in the parentheses. The `delay()` function takes a single argument and its function prototype looks like this:

```
void delay(unsigned long ms);
```

Recall that a **unsigned long** is a 32 bit unsigned value that we would call a `uint32_t`. This means that the range of values that can be passed to the `delay ()` is between 0ms and 4,294,967,296ms (or about 49.7 days).

If we wanted to create a function that take two arguments, that will flash an led pattern t times and has a delay of ms milliseconds it may look something like this:

```
void lightshow(uint8_t t, uint32_t ms)
{
    while( t > 0 )
    {
        digitalWrite(0,HIGH);
        delay(ms);
        digitalWrite(0,LOW);
        digitalWrite(1,HIGH);
        delay(ms);
        digitalWrite(1,LOW);
        digitalWrite(2,HIGH);
        delay(ms);
        digitalWrite(2,LOW);
        digitalWrite(3,HIGH);
        delay(ms);
        digitalWrite(3,LOW);
        t = t - 1;
    }
}
```

Return Value

In addition to being able to pass parameters to a function, it is also possible to get a single value back from a function. This is done by changing the void keyword prior to the function name to a datatype that we would like to return. The following example adds two uint32_t values together and returns the result using the return keyword.

```
uint32_t add(uint32_t value1, uint32_t value2)
{
    return (value1 + value2);
}
```

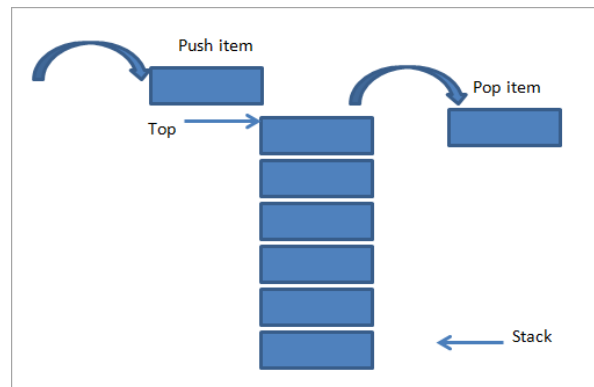
The astute observer may have already come to the conclusion that it seems very limiting to only be able to return a single value from a function, and indeed it is. When we get to discussion parsers and pointers in future sections we will demonstrate how to get around these limitations.

The Call Stack

Our programs exist in the memory of the computer as machine code. The size of the memory cell can vary from chip to chip, but in each case each memory cell has a unique address. When a program is running it starts at a low address and advances instruction by instruction. A register inside the microcontroller called a program counter (or PC) keeps track of the next instruction to execute. When we call a function we are tell the microprocessor to update the program counter to the location of the new function, but if we want to get back to where we came from then we must keep track of the next address after the point in the program where we make the call. This is called the return address. The return address is stored in a reserved section of memory called the stack (or call stack). A stack is so named because information is stored on it like trays in a cafeteria. The tray (or address) are stacked for waiting customers. That is to say the first tray placed on the stack is the last tray removed or conversely the last tray placed on the stack is the first removed. This storage abstraction is sometimes known as LIFO (Last In First Out).

<https://godbolt.org/>

```
char add(char a, char b) {  
    return a + b;  
}  
char call()  
{  
    add(3, 4);  
}
```



Source: <https://www.softwaretestinghelp.com/stack-in-cpp/>

In addition to storing the return address, the stack also stores all the parameters that are to be passed to our function. The return value is not typically stored on the stack. The return value would be stored in either a register of the microcontroller or special memory reserved for return values and is dependent on the compiler and microprocessor architecture in use.

Introduction to Interrupts

An interrupt is a hardware event that *interrupts* the typical flow of our program. An interrupt can be triggered by many different peripherals in the microcontroller including but not limited to a change of logic state on a pin or a value being received on a serial port.

When an interrupt occurs, our program is suspended and its state saved on the stack of the microcontroller. The microcontroller then begins execution of the Interrupt Service Routine (ISR) that "services" the interrupt that was triggered. The ISR responds to the triggered event and then exits returning to normal program flow.

The ISR can be thought of as a function that is called not by our program but when the hardware event occurs. In this class most ISR's that we experience have been written and are contained in the core code.

Week 4 – Asynchronous Serial Communications

Class Goals

Serial Communications

Introduction

Asynchronous Serial Communications is a mouthful to the uninitiated, but it is such a common thing that surrounds and enriches all of our lives on a daily basis that it can probably be unequivocally stated that it is probably the most important thing to understand from the perspective of bettering one's self by the study of electronics technology.

To understand what is meant by Asynchronous Serial Communications and the details of its nature it is best to tap into our prior knowledge and then step back a bit and look at some historical examples that we are probably all familiar with.

Accessing Prior Knowledge

We should all intuitively know what is meant by communications. It is a fundamental to human nature to communicate with each other from human being to human being. We do it instinctively at birth and get better at it as we grow. On a daily basis, when we can't be with people, we wish to communicate with we use technology that has been developed over the millennia to fulfill this most human of needs.

History

So now that we have established the common ground general understanding of what is meant by communications, we can review some historical technologies that we should all be familiar with. These technologies by modern standards are passé, because they surround us and are fully integrated into our daily lives which makes us numb to their presents, but each provide a significant step to understanding Asynchronous Serial Communications and almost define it in terms of what we need to know to be able to successfully use it when working with modern computer systems that utilize it.

Before we start, a bit of qualification must be given. The author of this article is not a historian and the references and dates have only been cursorily checked, so statements may be a bit askew from accepted historical fact but are given to paint a picture for the purpose of study for the subject at hand. With that said the author believes the information is considered correct. Additionally the conclusions may not be perfect either but are used to paint a general picture of trends in technology.

30000 B.C. Cave Paintings

Based on widely accepted and scientifically verifiable evidence, homo sapiens (humans) first appeared on the earth around 200,000 B.C. and apparently, we as a species wondered about for some time.



Then about 30,000 B.C. it is known that someone was drawing pictures on cave walls in Europe (these were probably drawn by Neanderthals and not homo sapiens) in what is now France, clearly an attempt at some kind of communications. If not intentional, they left on the wall of that cave a message that let us today know what was on their mind 30,000 years ago. If you were to ask me, it looks like they were thinking about what to eat lunch.¹

3000 B.C. Written Word

This leads us to the second development we need to consider. The development of written language about 3000 B.C. Once this development was out of the bag, it allowed ideas to be **stored** and **transferred** from place to place and generation to generation like never before. Around 40 years later the great pyramid of Giza was built, though maybe not a direct affect of the written word, surly a contributor to the success of this monument.²



2400 B.C. Postal Systems

Once there was written word, it only took about 600 years before the realization that there could be great benefit of sending messages over long distances by means of a courier. Again taken for granted today since we are inundated with bills and advertisements, prior to the development of postal systems communications to be made in person, or by traveling to where the record was written (since they were usual written on the side of a wall).³

¹ Clottes, Jean. "Chauvet Cave (ca. 30,000 B.C.)." In Heilbrunn Timeline of Art History. New York: The Metropolitan Museum of Art, 2000-. http://www.metmuseum.org/toah/hd/chav/hd_chav.htm (October 2002)

² Mark, Joshua J. "Writing" <https://www.ancient.eu/writing/> (April 2011)

³ Bellis, Mary. "History of Mail and the Postal System." ThoughtCo, Feb. 11, 2020, thoughtco.com/history-of-mail-1992142.

1439 A.D. Printing Press

The next technology we look at is the development of the Gutenberg moveable type printing press in 1439.⁴ Although not significantly important for understanding Asynchronous Serial Communications it is certainly worth mentioning in the development of technologies that create significant changes in the way we communicate and forever change the world. Prior to the creation of the moveable type printing press, the great literary works of mankind had to be painstakingly copied by hand, a task which was so time consuming it restricted access to these works to those with the wealth to employ someone for the time it took to do the transcription. Not to mention that hand transcription can be riddled with errors.



The story of Uppercase and Lowercase

To appreciate how significant a change the movable type printing press was to our daily lives we illustrate some terminology that was created for use with the printing press that has made it into everyday language, while etymology is lost in its daily use. Prior to the creation of the movable type printing press letter were either capital or not (capital). When the printing press went into common use a wooden case was built to house the type. The case was divided into drawers of which the upper part of the case (the uppercase) housed the capital letters and the lower part of the case (the lowercase) housed all other letters. With the advent of digital printing, printing presses have all most all disappeared except for in museums yet the terminology remains.

1819 A.D. The relationship between electricity and magnetism

In 1819 Hans Christian Ørsted unified electricity and magnetism through the observations that a current flowing through a wire can cause a compass needle to deflect from its normal orientation.

1831 A.D. The electric motor

Michael Faraday (for whom we name the unit of measure of capacitance after) explored electro-magnetism flushing out the nature of the force. In the process of doing so created the world's first electric motor. Faraday along with James Clerk Maxwell and many other created the theories that we utilize on a daily basis to build our modern society.

⁴ First movable type printing press in western culture. Other similar eastern technologies predate the Gutenberg printing press. Something I learned while visiting the Gutenberg Museum in Mainz, Germany where Gutenberg was borne and lived.

1836 A.D. Mores Code

The printing press was surely a significant improvement in communication technology which allowed old ideas to be shared with vast audiences like the written word could never be before. Postal systems improved over time and the means to create written documents improved as well, but the written word is great for studying the ideas of the past, even if they are only a day old, they are still old. We as humans have a more immediate need to communicate and communicate over long distances and this need was first satisfied in 1836 with the creation of Mores Code and the telegraph. Although the concepts of Mores Code and the Telegraph are probably familiar to all of us, they merit further study in the understanding of Asynchronous Serial Communications since the telegraph is the first device that can be said to utilize such communications.

International Morse Code

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to seven dots.

A	• —	U	• • —
B	— • • •	V	• • — •
C	— • — •	W	— • — •
D	— • • •	X	— • • —
E	•	Y	— • — • •
F	• • — •	Z	— — • •
G	— • — •		
H	• • • •		
I	• •		
J	• — — —		
K	— • • •	1	• — — — —
L	— • • • •	2	• • — — —
M	— —	3	• • • — —
N	— •	4	• • • • —
O	— — —	5	• • • • •
P	• — — —	6	— • • • •
Q	— • — —	7	— • • • • •
R	— • • •	8	— • • • • • •
S	• • •	9	— — • • • •
T	—	0	— — — — • •

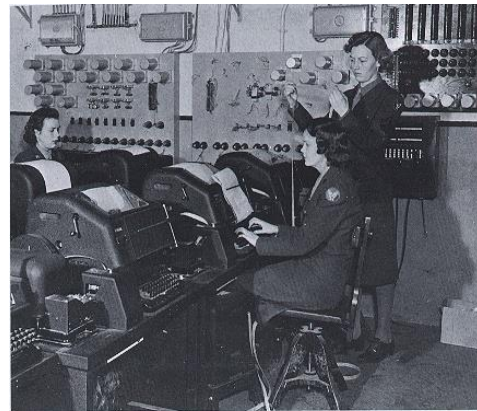
1886 A.D. Telephone

Alexander Gram Bell invented the telephone. This is noted here but reference but is slightly off the track of our topic of study for the moment.

1910 A.D. Teletype

A teleprinter (teletypewriter, Teletype or TTY for TeleTYpe/TeleTYpewriter) is a now largely obsolete electromechanical typewriter that can be used to communicate typed messages from point to point and point to multipoint over a variety of communications channels that range from a simple electrical connection, such as a pair of wires, to the use of radio and microwave as the transmission medium. - Wikipedia

The term TTY is still used in Linux/UNIX systems to describe the command line connect to the operating system.



WACs assigned to the Eighth Air Force in England operate teletype machines. (DOD photograph)

1973 A.D. Ethernet Invented

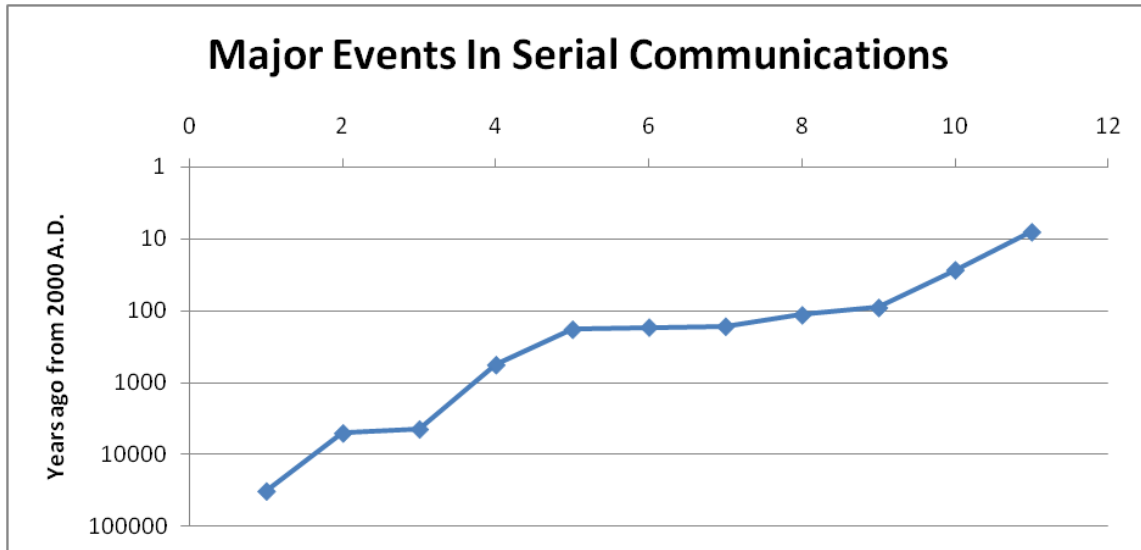
Multiple Inventors (Xerox PARC and Others).

1992 A.D. SMS Text Messaging

Multiple Inventors (Ericson Communications and Others).

A Trend to Note

The chart below graphs the events listed above on a logarithmic y-axis. The line is almost straight which implies non-linear growth. The trend here is that the development of technology speeds up the development of technology. It's a positive feedback loop. The loop is throttled by the human capacity to understand and synthesize new information; however, the limits of humanity are being pushed further with the advent of machine learning.



Where we find Serial Communications

The table below lists some places where serial communications can be found and the speeds and ranges of these technologies.

Technology	Speed	Range
SPI	MHz	Chip to Chip (on a PCB)
I2C	MHz	
RS-232	KHz	Short Range (Device to Device)
USB (Universal Serial Bus)	MHz-GHz	
FireWire	MHz	
MIDI (Musical Instrument Digital Interface)	KHz	
SATA (Serial ATA)	MHz	
Ethernet	MHz-GHz	Long Range (LAN)
RS-423	KHz-MHz	
RS-485	KHz-MHz	
DMX512 control of theatrical lighting	KHz	
Fibre Channel Connecting computers to mass storage	MHz-GHz	Very Long Range (WAN)
SONET and SDH Telecommunication over optical fibers	MHz-GHz	
T-1, E-1 and variants Telecommunication over copper pairs	MHz	
BlueTooth	KHz	
Zigbee	KHz	Wireless Short Range
Wireless USB	KHz-MHz	
WiFi	KHz-MHz	Wireless Mid Range
WiMax	MHz	Wireless Long Range

Asynchronous Serial Communications (Thirty Thousand Foot View)

Communications

The fundamental nature of computers and computation is that they are essentially information manipulation and transportation devices. The transportation of information is the communications part of the ASC (Asynchronous Serial Communications). Information manipulation takes place in the CPU.

Serial

Data in a computer is represented with bits. The transportation of bits takes place on conductors. Conductors can be arranged in two different schemes. Parallel or Serial.

Parallel Communications

With parallel communications more than one conductor is used (in parallel) to move as many bits as there are conductors per time splice.

Serial Communications

Serial communications achieves the same ends as parallel communications but instead of sending multiple bits over multiple conductors, the bits are all sent on a single conductor. Typically only a single

bit can be represented in single moment. To send multiple bit time is divided into several moments and a single bit at a time is sent one after another.

Tradeoffs

The tradeoff between serial and parallel is simply speed. For an equivalent transmission path, a parallel communications channel will always outperform a serial communications channel by the number of parallel conductors. So then serial communications are usually chosen for probably the common reason any decision is made in electronics, it is chosen because a single channel will cost less than multiple channels.

Asynchronous

Asynchronous means without synchronization. This implies that there probably exists somewhere in the world Synchronous Communications and indeed there does. A communications channel that is Synchronous in nature has a timing signal which makes it easier to detect signals, but the synchronization requires another conductor for the clock which again adds cost to a system. The ability to detect the bits being transmitted is achieved by the sender and the receiver agreeing ahead of time how long (in time) each bit will be presented before changing to the next bit (this is the asynchronous part).

Duplex

In order for communications of a message to take place it is required for there to be at least one originator (sender) and one destination (receiver), without which no communications can take place. Special names have been given to represent how communications networks are constructed so that engineers and technicians can speak about configurations without ambiguity while working.

Simplex

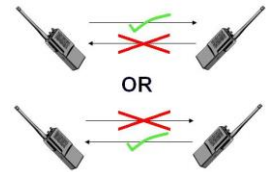
In simplex communications, information travels in a single direction at a time using a single communications line.

ADVANCED UNDERSTANDING

Advanced communications systems use a technique called clock recovery that allows a local clock to be synchronized to the transitions coming from incoming data that can be used as a clock signal. The downside of this scheme is that some data will be lost until the local clock is synchronized. Also if long periods of time elapse between transmissions the local clock can become unsynchronized, to get around this short coming data is encoded in such a way that the data line is always changing and the ones and zeros are represented as either a phase shift or lack of phase shift.

Half Duplex

In half duplex communications, information travels in two directions but only in one direction at a time, single or dual communications lines can be used. This communication duplex is similar to how walkie talkies work by using a single carrier frequency to transmit voice communications. Only one transmitter is possible at a time. This is not a great analogy since there can be multiple receivers. When we use half duplex in a computer system there is one transmitter and one receiver.



Full Duplex

In full duplex communications, information travels in two directions simultaneously, dual communications lines are required. Think of telephone communications where both parties can talk and listen at the same time.



Broadcast

Though not typically associated with ASC but used is broadcasting. If you have a single sender and many receivers this would be a broadcast type of topology.

RS-232

RS-232 is a voltage standard for digital ASC. RS-232 actually stands for Recommended Standard number 232 and is now maintained by the EIA (Electronica Industry Association) and sometimes called EIA-232. You can think of RS-232 as a voltage standard used in communications systems similar to how you might think of TTL (Transistor Transistor Logic) as a voltage standard used in digital electronics (Such as is used in MtsAC ELEC 56 classes). For many years, RS-232 stood as the defacto standard for ASC. This standard has only recently been displaced (starting around the year 2006) in computer systems by USB. Communications to Fubarino's and Arduino's utilize the USB CDC (Communications Device Class) which emulate the ideas from RS-232 over USB.

Necessary Conditions for RS-232 Communications

As mentioned previously, ASC relies on the sender and receiver to pre-agreed upon the parameters of the communication ahead of time. If there is disagreement, this will likely result is the receiver misinterpreting incoming data as gibberish. To make sure gibberish is not received some necessary conditions need to be agreed upon before communications can occur.

Bit rate (Baud rate)

Bit rate is the parameter used to defined for the sender and receiver how time is to be divided when using ASC. Bit rate calculations should look familiar to those with some electronics background (MtsAC ELEC 50B) since the formulas are equivalent to frequency and period relationships.

$$\frac{\text{bit}}{s} = \frac{1}{s/\text{bit}}$$

or bps (bit per second) = 1 / spb (seconds per bit) or the period

Bit rate is sometimes called baud rate but the two terms are slightly different. Bit rate refers to the number of bits per second that can be transmitted, while the baud rate refers to bandwidth of the communications channel. For example, if you are transmitting 10,000 bits per second, but the

communications channel only has a band width of 2,500 Hz, then a compression scheme has been used to stuff more bits onto a single cycle of communications thus the channel has baud rate of 2,500. Some methods used for stuffing more bits per cycle are known as frequency shift keying (FSK), phase shift keying (PSK), amplitude shift keying (ASK), some combination of all three or yet possibly something altogether different. Often the bit and rate and baud rate are the same and as a result the term baud rate is often improperly used to describe the bit rate.

Mark and Space

The logic of bits is inverted in the RS-232 standard from what we are used to looking at. Normally for 5V TTL or 3V LVTTTL (Low Voltage Transistor Transistor Logic) a logic one bit is a positive voltage and a zero bit is a voltage near zero. In RS-232 a one bit is a negative voltage and a zero bit is a positive voltage. This can cause confusion so often the bit levels are talked about in terms of a mark (one) or a space (zero) rather than high or low. The following chart summarizes:

Logic				Binary	TTL	LVTTTL	RS-232
Low	FALSE	Off	Space	0	0V	0V	+V
High	TRUE	On	Mark	1	5V	3.3V	-V

Remember for TTL, LVTTTL and RS-232 as well, logic level are not specific voltages but ranges which might look something like is shown in the table below. These ranges vary from manufacture to manufacture and chip to chip. Its best to consult the datasheet for any given chip you are using to know for sure the correct voltage range and to understand if the output of one chip can drive the input of another.

Logic Level	Voltage Rang
0	0V – 0.8V
Undefined	0.8V -2.0V
1	2.0V – 5V

Framing (Start bit, Stop bits and bit ordering)

When data is transmitted over an ASC RS-232 line it is sent a bit at a time (usually in groups of eight). These bits are framed with a start and stop bits. There is typically always one start bit, but the number of stop bits can be one or greater. Common values for stop bits are 1, 1.5 and 2. The start bit is always low (space) and the stop bits are always high (mark). When data is not being transmitted the line is said to be idle. An idle line is at the mark (logic 1) level.

Bit Order

Data is transmitted least significant bit (LSB) to most significant bit (MSB) vs time. Below are several examples of an ASC data transmission.

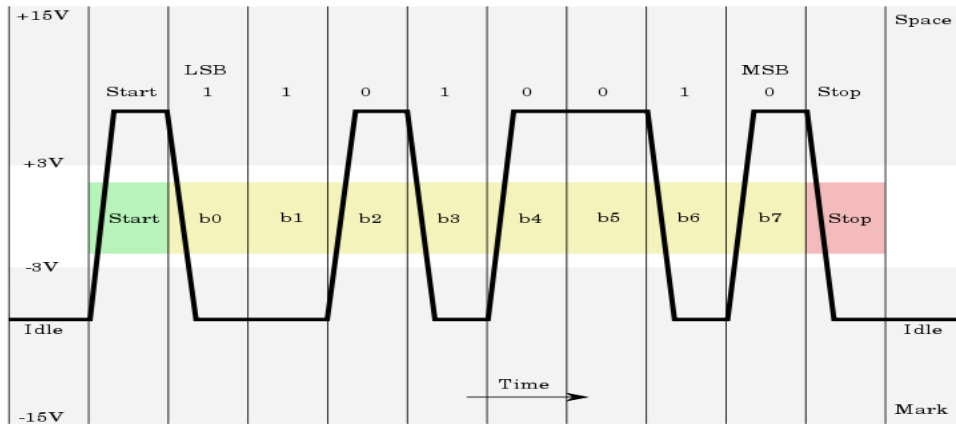


Figure 1 – RS-232 asynchronous serial communication frame (0x4B) From Wikipedia

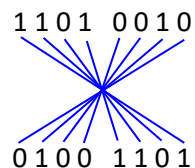
If you read the bits off the above chart from left to right, we see the bit sequence (remember logic 0 is a positive voltage):

0 1 1 0 1 0 0 1 0 1

The first bit and last bit are the framing bits start and stop. If we remove the framing bit, we are left with:

~~0~~ 1 1 0 1 0 0 1 0 ~~1~~

To interpret these values, we need to reverse their order from LSB to MSB to MSB to LSB:



0x4B (ASCII character 'K')

Transmission of the character 'A' (ASCII 0x41)

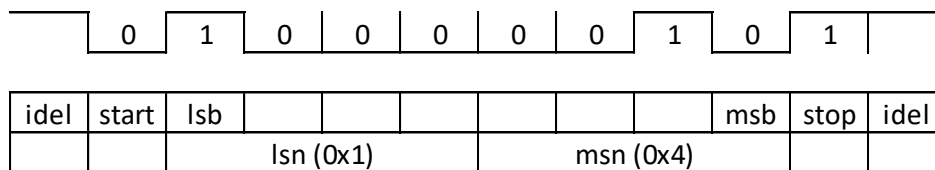


Figure 2 – Logic Level diagram Idle is spelled incorrectly (idel) in this graphic

Error

The nature of ASC means that the receiver and the transmitter must agree ahead of time on the bit rate of transmission. The bit rate is generated by an on the on-board clock. For the boards used in this class the clock used is derived from the onboard crystal oscillator for the microcontroller. Because sender and receiver have different clocks and clocks have a frequency a tolerance (or an operating band of frequencies) the clock frequency between the sender and receiver will not be the same. The difference between the sender clock and the receiver clock is call the error. The error from the clock sources means that there will be slight differences in the width of a bit from one system to another. When designing a system must be verified that error between clock sources is within acceptable limits in order for reliable communications to occur.

If we start with a frame that has 10 bits and there is a 10% error in clock speeds (and thus a 10% error for the length of time the receiver expect a bit to be vs what it actually is) this will mean that by the time the 10th bit is transmitted that the error will have added up to 100% (The error adds with each bit transmitted to 10% x 10 bits = 100%). So clearly 10% error is unacceptable. An error of 1% per bit will mean a total error of 10% per 10-bit frame, which will probably work fine, but the lower the error the better. Because the start bit indicates the beginning of a frame the clock of the receiver is resynchronized with the start of each new frame and the error is zero at this point.

$$\text{Error Per Bit\%} = (\text{Sender Bit Period} - \text{Receiver Bit Period}) / \text{Receiver Bit Period} \cdot 100\%$$

Break

Though not commonly used, it is important to understand the break signal.

A break is generated by holding the transmit line of the ASC system at space (logic low) for longer than the frame length. This allows the receiver to detect 257 different values (256 from the 8 data bits and 1 for the break). This could be useful if you are trying to send 8-bit binary data and need some way to signal the receiver that some exceptional condition has occurred.

Parity Bit

Another not often used parameter of ASC is called a parity bit. Early systems used parity bits as a method of detecting errors in communications. The way a parity bit works is for an 8-bit data word a 9th parity bit is added to the frame. The parity bit is set so that the count the one bits being transmitted (including the parity bit) and result in a sum that will be even (known as even parity) or an odd (known as odd parity). For example if odd parity is selected and transmission of the data 0b 1001 0110 is to occur, there are four one's, four is an even number so the parity bit will be set to one (0b 1001 0110 1) making count of one bits equal to five, which is an odd number. The same number transmitted using even parity would have the parity bit set to zero (0b 1001 0110 0). Parity bit is not used much anymore because it just is not very good at catching errors and there are much better mechanisms have been developed that require less communication bandwidth. One such better method is known as a CRC (Cyclic Redundancy Check). A CRC works by using a register to accumulate value that generated by passing the data to be transmitted through the CRC algorithm on the sender's computer. Then the receiver does the same with the received data. The sender then sends is CRC value it calculated and the

receiver compares its received CRC value to the its own calculated value. If CRC values match the data is assumed good. If not the receiver request the same data to be transmitted again.

PC Communications Notation

MS-DOS and Windows computers made popular a notation for representing how an asynchronous communications port is set up. The notation looks like this:

COM1: 9600,8,N,1

COM1 indicates the physical PC COMmunications port.

9600 is the bits per second.

8 is the data bits per frame.

N is the parity (N=None, E=Even, O=Odd).

1 is the number of stop bits in the frame.

Each frame always has 1 start bit.

Representing Information as Digital Data

ASCII (American Standard Code for Information Interchange)

ASCII is a seven bit code that is used to represent alpha numeric data as well as some special characters and is extremely prevalent in computing probably for no other reason than due to the length of time it has been around.

The following chart can be used to lookup an ASCII character from a hexadecimal ASCII code or visa versa.

HI\LO	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

To find the ASCII code for the ASCII character 'A' (capital A), first find the character 'A' in the chart then combine first the hex nibble to the left chart followed by the hex nibble from the top of the chart. The ASCII code for 'A' is 0x41. Conversely, if we have the ASCII Code 0x38 we use the high nibble (3) to find the row on the chart and the low nibble (8) to find the column. The ASCII character for the ASCII code 0x38 is the character '8'.

Some of the characters in the ASCII chart are known as “whitespace” or non-printable because they do not render to a visible character. Examples of these are SP (Space) and HT (Horizontal Tab). A list of non-printable characters and their descriptions are shown below.

Symbol	Description	Symbol	Description
NULL	Null character	DC1	Device Control 1
SOH	Start Of Header	DC2	Device Control 2
STX	Start of Text	DC3	Device Control 3
ETX	End of Text	DC4	Device Control 4
EOT	End Of Transmission	NAK	Negative Acknowledge
ENQ	Enquiry	SYN	Synchronous Idle
ACK	Acknowledge	ETB	End of Transmission Block
BEL	Bell (Ring a bell)	CAN	Cancel
BS	Backspace	EM	End of Medium
HT	Horizontal Tab (Tab)	SUB	Substitute
LF	Linefeed	ESC	Escape
VT	Vertical Tab	FS	File Seperator
FF	Formfeed	GS	Group Seperator
CR	Carriage Return (Enter)	RS	Record Seperator
SO	Shift Out	US	Unit Seperator
SI	Shift In	SP	Space (Space Bar)
DLE	Data Line Escape	DEL	Delete

Terminology

DTE

Data Terminal Equipment (The Computer)

DCE

Data Communications Equipments (The Modem)

Configuring the UART of a chipKIT or Arduino board for ASC

The complexity of configuration of the UART for chipKIT or Arduino board is taken care of by the core libraries of the system. All that we need do is specify the bit rate by using the Serial.begin(speed) function. This is typically called from the setup() but need not be called there alone.

If your target board has more than one serial UART additional serial ports can be configured using the SerialX.begin(speed) command where X is a number that represents the serial port on the board.

Demo

Iterating through a string

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  delay(3000);
  char class_name[] = "MtSAC ELEC 74";
  uint8_t i;
  i = 0;
  while (i < sizeof(class_name)) {
    Serial.print("class_name[");
    Serial.print(i, DEC);
    Serial.print("] = ");
    Serial.print(class_name[i], DEC);
    Serial.print(",");
    Serial.print(class_name[i]);
    Serial.println(".");
    delay(1000);
    i = i + 1;
  }
  Serial.println(".");
  Serial.println(".");
}

void loop() {}
```

View the output in HEX by changing this:

```
Serial.println(class_name [i], DEC);
```

to this:

```
Serial.println(class_name [i], HEX);
```

To see one past the end of the string change the will condition to this:

```
while(i <= sizeof(class_name))
```

The end of a string is marked by the NULL character. The NULL is the numerical value zero. We can use the fact the strings are terminated with a NULL to look for the end of the string by changing the while loop to the following:

```
while(class_name [i] != 0)
```

Next we pull the while loop out of the setup and create an examine function. The program passes the string from the setup() to the examine() by pointer. Accessing an individual element of a string is done

using the index brackets '[' and ']'. To access the address in memory, and hence the entire string, you just need to drop the index brackets. The complete code is given below:

```
void examine(char *p)
{
    int8_t i;
    i = 0;
    while(p[i] != 0 ) {
        Serial.print("p[");
        Serial.print(i, DEC);
        Serial.print("] = ");
        Serial.print(p[i], HEX);
        Serial.print(",");
        Serial.print(p[i]);
        Serial.println(".");
        delay(500);
        i = i + 1;
    }
}

void setup() {
    Serial.begin(115200);
    delay(3000);
    char class_name[] = "MtSAC";
    examine(class_name);
}

void loop() {
    // put your main code here, to run repeatedly:
}
```

If you want to reverse the string you need to know how long it is and start by printing the last character first. In order to do this you need to find out how long the string is and print it last character first. You can calculate the length of a string with a function like this:

```
int32_t length(char *p)
{
    int32_t i = 0;
    while(p[i] != 0 ) {
        i = i + 1;
    }
    return i;
}
```

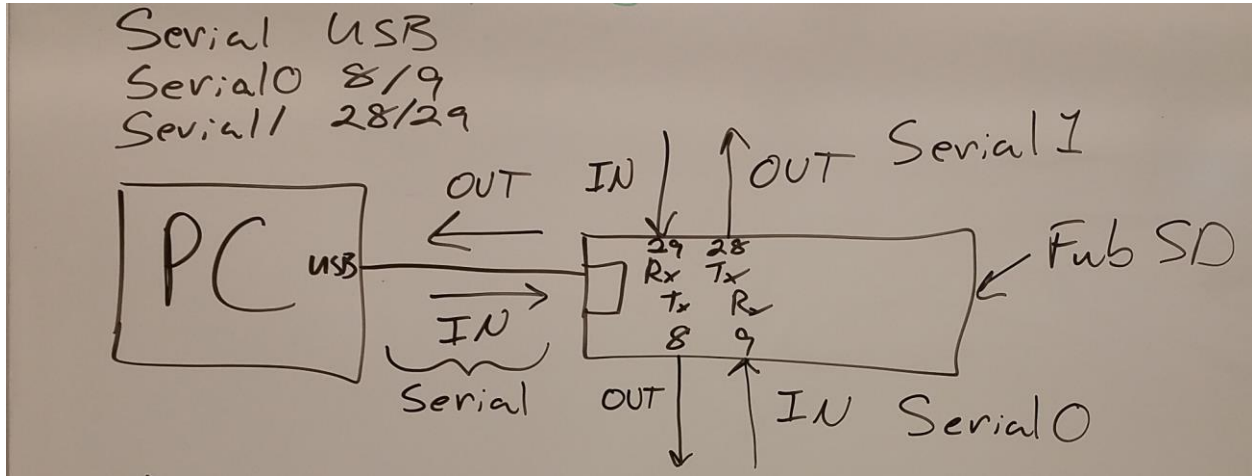
You would use it like this:

```
void examine(char *p)
{
    uint8_t i;
    i = length(p);
    while(i >= 0 ) {
        Serial.print("p[");
        Serial.print(i, DEC);
        Serial.print("] = ");
        Serial.print(p[i], HEX);
        Serial.print(",");
        Serial.print(p[i]);
        Serial.println(".");
        delay(500);
        i = i - 1;
    }
}
```

There is a built in function called `strlen()` that does the same thing as `length` above. We can use this like this:

```
void examine(char *p)
{
    uint8_t i;
    i = strlen(p);
    while(i >= 0 ) {
        Serial.print("p[");
        Serial.print(i, DEC);
        Serial.print("] = ");
        Serial.print(p[i], HEX);
        Serial.print(",");
        Serial.print(p[i]);
        Serial.println(".");
        delay(500);
        i = i - 1;
    }
}
```


Fubarino SD:



Fubarino Mini:

Serial: USB
 Serial0 Pins TX1/RX1 or 17/18
 Serial1 Pins TX2/RX2 or 26/25

Category	Serial Port Functions
Initialization	<code>Serial.begin(bps);</code> // Where bps = bits per second (for UART's)
Input	<code>Serial.available();</code> // Returns the number of bytes available to read 0=False; 1+ True <code>Serial.read();</code> // Read a single byte of data
Output	<code>Serial.write(c);</code> // Write a single byte of data <code>Serial.print(p);</code> // Formatted data output <code>Serial.println(p);</code> // Formatted data output + newline characters "\r\n"

<https://www.arduino.cc/reference/en/language/functions/communication/serial/>

Week 5/6 – Pointers, String Functions

Review

From: <https://www.arduino.cc/reference/en/language/functions/communication/serial/>

Serial.begin()

Description

Sets the data rate in bits per second (baud) for serial data transmission. For communicating with Serial Monitor, make sure to use one of the baud rates listed in the menu at the bottom right corner of its screen. You can, however, specify other rates - for example, to communicate over pins 0 and 1 with a component that requires a particular baud rate.

An optional second argument configures the data, parity, and stop bits. The default is 8 data bits, no parity, one stop bit.

Syntax

```
Serial.begin(speed)
```

```
Serial.begin(speed, config)
```

Parameters

Serial: serial port object. See the list of available serial ports for each board on the Serial main page.

speed: in bits per second (baud) - long

config: sets data, parity, and stop bits. The default value is: SERIAL_8N1

Serial.available() - Input

Description

Get the number of bytes (characters) available for reading from the serial port. This is data that's already arrived and stored in the serial receive buffer (which holds 64 bytes).

Example:

```
if(Serial.availble() > 0)
{
    // We received some data
}
```

Serial.read() - Input

Description

Reads incoming serial data.

Example:

```
if(Serial.available() > 0)
{
    char c = Serial.Read();
}
```

Serial.print() - Output

Description

Prints data to the serial port as human-readable ASCII text. This command can take many forms. Numbers are printed using an ASCII character for each digit. Floats are similarly printed as ASCII digits, defaulting to two decimal places. Bytes are sent as a single character. Characters and strings are sent as is. For example-

```
Serial.print(78) gives "78"
Serial.print(1.23456) gives "1.23"
Serial.print('N') gives "N"
Serial.print("Hello world.") gives "Hello world."
```

An optional second parameter specifies the base (format) to use; permitted values are BIN(binary, or base 2), OCT(octal, or base 8), DEC(decimal, or base 10), HEX(hexadecimal, or base 16). For floating point numbers, this parameter specifies the number of decimal places to use. For example-

```
Serial.print(78, BIN) gives "1001110"
Serial.print(78, OCT) gives "116"
Serial.print(78, DEC) gives "78"
Serial.print(78, HEX) gives "4E"
Serial.print(1.23456, 0) gives "1"
Serial.print(1.23456, 2) gives "1.23"
Serial.print(1.23456, 4) gives "1.2346"
```

You can pass flash-memory based strings to Serial.print() by wrapping them with F(). For example:

```
Serial.print(F("Hello World"))
```

To send data without conversion to its representation as characters, use Serial.write().

Syntax

```
Serial.print(val)
```

```
Serial.print(val, format)
```

Parameters

Serial: serial port object. See the list of available serial ports for each board on the Serial main page.

val: the value to print - any data type

Returns

size_t: print() returns the number of bytes written, though reading that number is optional.

Serial.println() - Output

Description

Same as Serial.print() but adds a newline at the end of what is being printed.

Class Goals

In order to process the characters being received from the USART in an interrupt, we will need to develop some more C language skills and gain some further insight into how memory is use in the C programming language.

The big picture: When a character is received by the hardware that is associated with a Serial class the PIC32 will generate an interrupt. In the ISR (Interrupt Service Routine) the received character is put into a buffer. A buffer is just a block of memory that will hold these received characters until they can be processed by our main loop. Then when the main loop has time to process the data that has come in through the USART it can act upon it.

Scope

The term “scope” is use often in programing to consider when a variable can be accessed by a certain piece of code. It is important to understand the scope of a variable so that you know if the data it stores can be manipulate by a specific piece of code.

Memory and Auto Variables

When we create an auto variable in the C programming language memory in our computer is allocated. Allocation is the act of deciding that this memory is used for a specific purpose, namely to store data for our newly created variable. When we are done with memory it is de-allocated, this happens automatically in C. Auto variables are in scope within the function where they are defined.

Global Variables

Global variables are variables that are accessible by all functions (hence the name global) and typically allocated on the heap rather than on the stack. Auto variables are rather temporary, but since globals stick around the entire time the program is running the heap is a better place for them since they would just take up space on the stack that is never released. Global variable are in scope everywhere within a program but they can be masked by an auto variable with the same name (demo to follow).

To declare a global, you make the declaration outside of a function and usually at the top of your c code file. The declaration is just like any other you have seen, just outside of a function. Here is an example of a program with a global variable:

ADVANCED UNDERSTANDING

All variables up until this point have been auto variables. That is to say memory for them is allocated automatically when we declared. These variables are allocated on the stack of the microcontroller. Stack size is often limited and there is another place we can allocate memory, this is called the heap. To allocate memory on the heap a command called alloc is used that will return a pointer to a chunk of the heap. There is no physical difference between the stack and the heap, they are just ideas that are used for organizing memory in the computer. There is, however, a difference in how the memory is organized.

```
// Global Variables
uint8_t global;

// Program
void setup()
{
  Serial.begin(115200);
  delay(3000);
  uint8_t local;
  local = 1;
  global = local;
  Serial.print("before foo: ");
  Serial.print(local, DEC);
  Serial.print(", ");
  Serial.print(global, DEC);
  Serial.println(".");
  foo();
  Serial.print("after foo: ");
  Serial.print(local, DEC);
  Serial.print(", ");
  Serial.print(global, DEC);
  Serial.println(".");
  Serial.println("This is a bunch of data.");
}

void foo(void)
{
  uint8_t local;
  local = 2;
  global = local;
}

void loop() {}
```

Notice that variable `global` is accessible in both `setup()` and `foo()` without passing it to the function. It is generally not a good practice of using a global variable since issues arise when you have a global and a local variable that have the same name, in particular, the local variable has precedence over the global and the global becomes inaccessible. Also, when you're looking at a more complex program it may be difficult to tell if you're looking at a global or a local variable.

```
// Global Variables
uint8_t ambiguous = 1;

// Program
void setup()
{
  Serial.begin(115200);
  delay(3000);
  uint8_t ambiguous;
  ambiguous = 2; // global variable masked by local
  Serial.print("ambiguous: ");
  Serial.print(ambiguous, DEC);
  Serial.println(".");
  foo();
  Serial.println("This is a bunch of data.");
}

void foo()
{
  Serial.print("ambiguous: ");
  Serial.print(ambiguous, DEC);
  Serial.println(".");
}

void loop() {}
```

So if globals are so bad, then why tell us about them? If you need to get information into or out of an interrupt you must use a global variable since you can't pass or return information to or from an interrupt.

So our buffer to store incoming characters from the serial port interrupt will be stored in a global variable.

Volatile Data Types

In order to use our global variables in both an interrupt and out of the interrupt we must declare them as volatile. The volatile keyword tells the C compiler that the memory could be operated on by outside functions and to make no assumptions about what values may be in the memory.

How can a C compiler make assumptions about what is in the memory of the computer? When a program is compiled data in memory may be copied into memory inside the microprocessor (typically called an accumulator or arithmetic register), if the compiler has copied a memory value into the accumulator and that memory changes by an interrupt then it may be different the next time it needs the value, so rather than referring to the copy in the accumulator it must re-read the value from memory.

Static Data Types

Declaring a local variable as static puts it on the heap and makes it so that it will retain its value between calls to the function. The variable only has scope within the function where it is declared.

Scope of a Variable

In programming, we say a variable is in scope if it is accessible in a certain area of the program. Global variable have global scope and local variable are only in scope within the function they are declared in.

Character (ASCII) Values

We know that in the microcontroller all data is binary. But we have seen that when we have an 8-bit data type we can either type directly in binary or have the compiler convert numerical values to binary for us. So the following lines of code are all equivalent and all store 00110101 into the variable x.

```
x = 0b00110101; // binary
x = 0x35;       // hexadecimal
x = 065;        // octal
x = 53;         // decimal
```

In addition to these convenient compiler features, the compiler will also convert an ASCII character to its numerical value for us. This is done by encapsulating the ASCII character we want the numerical value of in single quotes as such:

```
x = '5'          // ASCII 5 is 0x35 in hex
```

There exist some special escaped ASCII characters that cannot be represented in a text editor. These characters are often called non-printable or whitespace characters. In order to use these characters in your C programs an escape character followed by a printable character are used to represent these symbols. For example:

```
x = '\t'          // ASCII <TAB> is 9 or 0x09 in hex
x = '\n'          // ASCII <LF> is 10 or 0x0A in hex
x = '\r'          // ASCII <CR> is 13 or 0x0D in hex
x = '\0'          // ASCII <NULL> is 0 or 0x00 in hex
```

Even though two printable characters are used only a single ASCII value is stored in memory.

Strings Arrays

We have seen that we can make an array of continuous memory that is addressable with an index such as this:

```
uint8_t x[] = { 0x50, 0x49, 0x43, 0x20, 0x33, 0x32, 0x00 };
```

It is also possible to store ASCII values like this:

```
uint8_t x[] = { 'P','I','C',' ','3','2',0 };
```

In C there is a concept called a string array that allows us to more efficiently initialize the variable x to a name. This looks like this:

```
uint8_t x[] = "PIC 32";
```

All three of these statements do exactly the same thing. They allocates seven bytes, one for each letter in the string "PIC 32" and the seventh for the value zero that is stored after the '2' in memory. When working with strings the final zero is called the null terminator.

MEMORY RECALL

The datatype `uint8_t` is just a macro that is used to represent the C type `char`.

The array is placed in memory it looks like this:

x Array Offset	ASCII Value	C char Value	Hex Value
x[0]	P	'P'	0x50
x[1]	I	'I'	0x49
x[2]	C	'C'	0x43
x[3]	<SPACE>	' '	0x20
x[4]	3	'3'	0x33
x[5]	2	'2'	0x32
x[6]	<NULL>	'\0'	0x00

Each letter in the string is addressable by using the index. Where x[0] is 'P' and x[4] is '3' and x[6] is 0.

Null Terminated Strings

The reason the last value in a string is null is so that if we are looking through memory we don't have to know how many bytes (or characters) are stored in our string but rather just look for the null character.

Allocating strings (additional examples)

Most of the time in C, you will see strings using the `char` data type, recall that we have been using `uint8_t`, but most people are not as smart as us and use `char's`.

```
// allocate 7 bytes indexed from 0 to 6
// where string[0]= '1', string[5]= '\r' and string[6]=0 (or '\0').
char string1[] ="12345\r";
```

```
// allocates 10 bytes indexed from 0 to 9.
// values undefined (un-initialized)
char string2[10];
```

Pointers

A pointer is a special type of variable in the C programming language that holds the address of a memory location. Recall that the PIC32 has 32 bit memory range of 0x00000000 through 0xFFFFFFFF. This would mean that a pointer needs to store 32 bit value for this processor. In this book we will assume pointers are 32 bits in length.

A pointer can point to any memory location but are typically used to point to very specific things such as variable or a function. Although pointers to functions are very useful and allow for very dynamic and interesting code, in this book, we are just going to explore pointers to variables.

Every memory location in the computer has a physical numerical address. Very detailed information regarding the mapping memory is provided in the "Memory Organization" section (3) of the PIC32 datasheet.

An interesting thing to note is that the special function registers (SFRs) are mapped into the data memory of the PIC. When you look at "Memory Organization" section of the datasheet you can the physical address for each of the SFRs for the PIC.

ADVANCED UNDERSTANDING

The size of a pointer in for a PIC is configurable by issuing commands to the compiler and linker. This is done for two reasons. Some PIC's have more than 64K of program space and thus need a larger than 16 bit pointer. But this means we will use much more RAM for our pointers. And since memory in a PIC is so limited if we don't need to use this extra RAM we make our pointers smaller. The details of how this works can be found in your compiler manuals. In this book we assume addresses are 16 bits.

Declaring Pointers to Variables in C

Recall that we can simply create a variable called v that holds an unsigned eight bit value by doing this:

```
uint8_t v;
```

To create a "pointer" variable that stores an address to an unsigned eight bit value called pv we do this:

```
uint8_t* pv;
```

The difference between the first and second declaration is the addition of the '*' character when creating a the variable. When used here the '*' is what makes this variable a pointer and we would say that pv is a "uint8_t pointer".

Wait a minute here, I thought you said address need to be 32-bits on a PIC32 but you made pv a uint8_t pointer. Well, actually, pv does store a 32 bit address. And yes, all pointers on the PIC32 are 32 bits. The type "uint8_t*" just means that we intend to use the address stored in pv to point to the memory location of a variable that is of type uint8_t.

Obtain a Variables Address

To get the address of a non-pointer variable prepend the non-pointer variable with the ampersand character as such:

```
uint8_t v;
uint8_t* pv;
v = 1; // Assign the number 1 to v
pv = &v; // Assign the address of v to the
        // pointer pv (pv now points to v)
```

Dereference a Pointer

A pointer is a variable that lets you reference another variable by its address. To obtaining the value that is pointed to by this variable you need to dereferencing the pointer. To dereference a pointer the pointer variable is prepended with an asterisk.

```
uint8_t v;
uint8_t* pv;
v = 1; // Assign 1 to v
pv = &v; // Assign the address of v to the
        // pointer pv (pv now points to v)
*pv = *pv + 1; // Add 1 to the value pointed to
              // by pv then store the result
              // back into the memory location
              // pointed to by pv (v = v + 1)
```

Using Pointers in C

So how do we use pointers? Let look at the following code:

```
uint8_t v;
uint8_t w;
uint8_t* pv;
uint8_t* pw;
v = 1; // Assign 1 to v
w = v; // Assign the value of v to w (w now equals 1)
pv = &v; // Assign the address of v to the pointer pv (pv now points to v)
w = *pv; // Assign the value pointed to by pv to w (pv point to v to w = 1)
pw = pv; // Assign the value of pv to pw (pw now also points to V)
*pv = *pv + 1; // increment the values pointed to by pv (v = v + 1)
v = *pw; // Assign the value pointed to by pw to v (pw points to v so v = v)
w = *pw; // Assign the value pointed to by pw to w (pw points to v so w = v)
(*pw)++; // Post increment the value pointed to by pw (v++)
pw++; // Post increment pw (pw now points to w)
*pw = *pv; // Assign the value pointed to by pv to the value pointed to by pw
        // (w = v)
```

CONFUSION POINT

When we speak of pointers in C there are two ideas that get intermingled. A pointer can be:

1. An address of a memory location.
2. A variable that holds an address to a memory location.

CONFUSION POINT

Dereferencing a pointer can be a bit confusing since the pointer itself is declared using the asterisk. The confusion comes from the asterisk having different meanings. When declaring the pointer it indicates that the variable is a pointer. When used in code, it indicates that the pointer is to be used a variable and not as an address.

Breakdown

Pointers can point to any variable type. Remember a type indicates how much memory is allocated and how its organized. So, `uint8_t` is a type that stores 8-bit and `uint16_t` is a type that stores 16-bits.

TYPE v;

v => evaluates to a value stored in memory of type TYPE

&v => evaluates to the memory address where the value v is stored

TYPE *pv;

*pv => evaluates to a value stored in memory pointed to by pv of type TYPE

pv => evaluates to a memory address where a value is stored (be careful, this is only true if you point it to something first)

Common Pit Fall with Pointers

The following example shows a common pit fall for those new to pointers. In this example the variable `p` is created but not initialized to point to any particular memory location. Then `p` is dereferenced and an attempt to store the value 32 in the memory location pointed to by `p`. Since we don't know where `p` points to, this could be disastrous for our program due to potentially overwriting something important we don't want to be overwritten.

```
void setup(void)
{
    uint8_t *p;
    *p = 32; // opps, where does p point to??
}
```

Another common pitfall is to return a pointer to a auto variable such as shown below:

```
char* foo(void)
{
    char x[] = "I only exist in this function";
    return x;
}

void setup(void)
{
    Serial.begin(115200);
    char *p;
    p = foo();
    Serial.println(p);
    Serial.println(p);
    Serial.println(p);
}
```

Since the variable `x` is an auto variable it is allocated on the stack when the function `foo` is called and deallocated when `foo` returns. Since `x` has been deallocated the returned pointer points to a portion of the stack that is sure to get overwritten the next time a function is called.

Pointers and String Arrays

If we have a string array such as:

```
char x[] = "PIC 32";
```

We can get access to the value of each location in the string by using an index in the square brackets. For instance, `x[0]` evaluates to the character 'P'.

But if we just evaluate `x` by itself, it returns the address of index zero.

```
char x[] = "PIC 32";  
char *pv;  
pv = x;
```

Stack

In this class we have already taken a look at one type of data structure called a stack. Recall that a stack stores data one on top of another with the principle characteristic that the first element put in the stack is the last out (FILO), and conversely, the last to element to be put in the stack is the first to come out (LIFO). Think of a stack of trays in a cafeteria, where the trays are the data that gets stored in memory and the stack is memory with the data stored in it.

We will not be implementing a stack, but since we have already seen them I thought it best to review before moving on due to the similarity to that of our next data structure.

Queue

Think of a queue data structure as a line of people waiting to use the bathroom. If you were British you might just say you were in the queue for the loo.

The queue is similar to the stack in that we say the first element in to the queue is the first out (FIFO) or the last element in to the queue is the last out (LIFO).

We will implement our serial receive buffer in a queue and we will implement our queue using arrays and index variables.

Under the hood (Serial Receive Interrupt)

Whenever a character is received by the target board an interrupt is generated that temporarily stops your program from running. The interrupt is then serviced which is to say the character received is read from the microcontroller register that contains the received character and it is put into a buffer. The interruption is quite short and unless we are doing something that requires strict timing requirements and we are looking at signals on an oscilloscope it will probably be very hard to detect the interruption and we will have the appearance that our program is uninterrupted.

The incoming stored data can be accessed by using the Wiring serial abstractions:

ADVANCED UNDERSTANDING

The way we will be implementing our queue is often called a circular buffer since we will be using the same memory over and over again in a circle.

`Serial.available()`

Which returns a value the number of bytes that have been received and are ready to be read.

`Serial.read()`

Which returns the next available character in the buffer.

Pointer Arithmetic

When you have a pointer variable, arithmetic operations can be done in the same fashion as non-pointer variable. When doing an arithmetic operation on a pointer do not dereference the variable.

```
char x[] = "PIC 32";
char c;
char *pv;
pv = x;
c = *pv;      // c = 'P'
pv = pv + 1;  // add 1 to pv
c = *pv;      // c = 'I'
pv = pv + 1;  // add 1 to pv
c = *pv;      // c = 'C'
pv = pv + 1;  // add 1 to pv
c = *pv;      // c = ' '
pv = pv + 1;  // add 1 to pv
c = *pv;      // c = '3'
pv = pv + 1;  // add 1 to pv
c = *pv;      // c = '2'
pv = pv + 1;  // add 1 to pv
c = *pv;      // c = 0 or '\0'
```

Pointers to strings

Standard String Functions

The following are a list of string functions that are defined in the `string.h` header included with boost and linked in from a library if used. They are shown using the C data type of `char*` and `int`. The string library and these functions are pretty common and can be found in most C compilers. If however you find they don't exist, you can recreate them simply.

*`unsigned char strlen(char *source)`*

Returns the length of a string (not counting the null termination).

*`void strcpy(char *destination, char *source)`*

Copies source to destination, memory for destination must already be allocated.

void strcat (char *destination, char *source)

Concatenates the source onto the end of the destination string, memory for destination must already be allocated.

signed char strcmp(char *s1, char *s2)

Compares two strings, if they are the same strcmp returns zero, if they are different strcmp returns a non zero value.

signed char strncmp(char *src1, char *src2, unsigned char len)

Compares the first n characters of two strings, if they are the same strcmp returns zero, if they are different strcmp returns a non zero value.

Sample implementation string pointer functions

int strlen(char *)

```
unsigned char strlen(char *s)
{
    int count = 0;
    while( *s != 0 ) {
        count++;
        s++;
    }
    return count;
}
```

Shown above is a possible implementation for the strlen function. The local variable count is used to store the number of characters found in the array. The loop exits when s points to a memory location containing the value 0x00. The string is iterated through by the use of pointer arithmetic and the loop completes when the value of 0x00 is found in the array. Importing point: if the array is not null terminated the function will never exit, or at least not until it finds a 0x00 in memory somewhere.

Custom String Functions

Besides the included string functions, we will often need to manipulate strings. The following are some string functions that we may want to use in our programs. Be sure you understand the mechanics of each function so that you can use what you have learned to write your own.

Sample implementation string pointer functions

`void toupper (char *)`

```
void toupper (char *s)
{
    while( *s != 0 ) {
        if( *s >= 'a' && *s <= 'z' )
            *s = *s - 0x20;
        s++; // increment the pointer by 1
    }
}
```

The `toupper()` take the address of a pointer to a null terminated character array and converts any characters that are between lowercase a and lowercase z (inclusive) by subtracting 0x20 to their ASCII value. The string is iterated through by the use of pointer arithmetic and the loop completes when the value of 0x00 is found in the array. Importing point: if the array is not null terminated the function will never exit, or at least not until it finds a 0x00 in memory somewhere.

`void tolower (char *)`

```
void tolower (char *s)
{
    while( *s != 0 ) {
        if( *s >= 'A' && *s <= 'Z' )
            *s = *s + 0x20;
        s++;
    }
}
```

The `tolower()` works exactly like `toupper()` with the exception that it converts any characters that are between uppercase A and uppercase Z (inclusive) by adding 0x20 to their ASCII value.

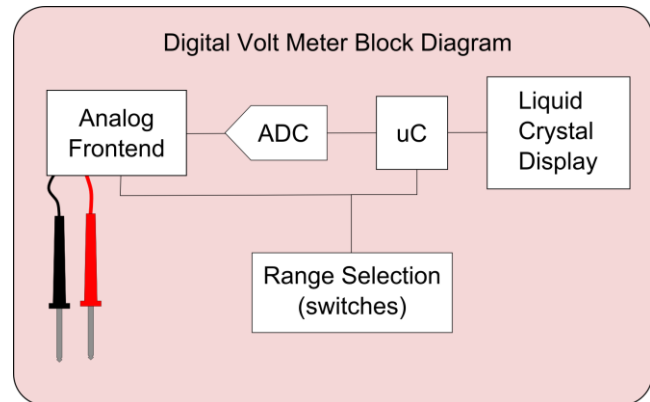
Week 7 – Analog to Digital Converter (ADC)

What is an ADC

An analog to digital converter (ADC) is just what its name implies, that is, it's a device that converts an analog voltage to a digital numeric value that represents the measured analog voltage. The ADC we will be using is built into the PIC microcontroller.

Example DVM

A device that we are all familiar with that has an ADC would be a Digital Volt Meter (DVM). In the image below the voltage that is to be detected is measured with the probes and passed through an analog frontend. Next the ADC converts the measured voltage to a digital number. The number is read from the ADC by the microcontroller then displayed on the liquid crystal display (LCD). The range selection switches configure the analog frontend so that the signal can be measured properly and let the microcontroller know what signal is being measured so that the software can set the appropriate display elements for user feedback as well as correctly scale values coming from the ADC into human readable format.

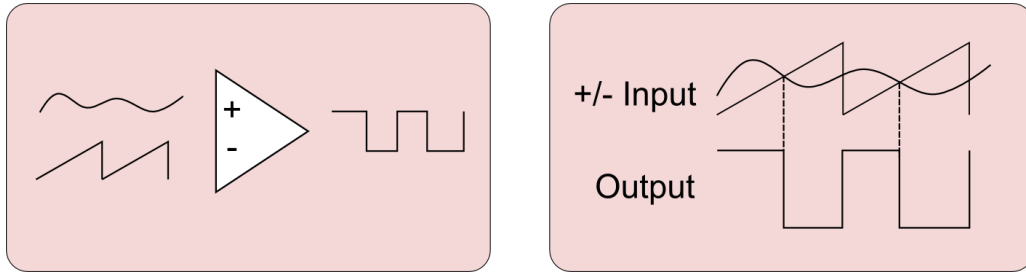


How does an ADC work

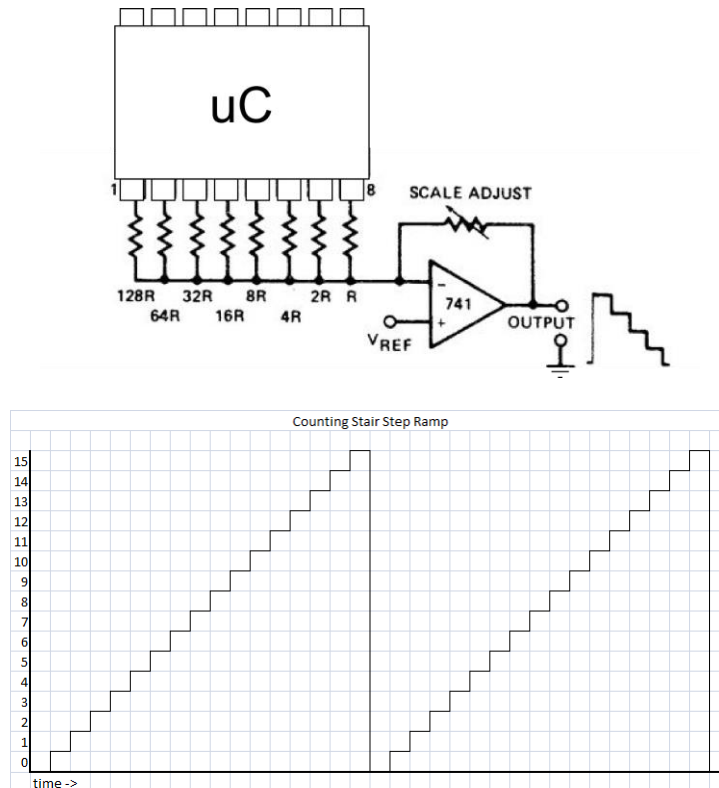
There are many different ways in which an ADC can be implemented. The implementations we will look at are for demonstration purpose to gain understanding of how the ADC works. They could be used as a basis for designing your own ADC, but from a practical standpoint an off the shelf ADC will be have a much more sophisticated implementation.

Counter and Comparator

At the simplest level an ADC compares two voltages using a comparator circuit. The two voltages compared are an unknown voltage and a known voltage. To find the value of the unknown voltage a known voltage is sweep from the ADC min to ADC max voltage. When the value of the known voltage becomes larger than the unknown voltage the output of the comparator changes and this change is digital and can be detected by a logic gate.



To generate the ramp voltage several digital outputs can be used to generate a current stair step signal that combined with a transimpedance amplifier (TIA) will create a voltage stair step. The circuit to generate a stair step might look something like this:



Classification

There are two primary attributes that classify the abilities of an ADC. Resolution (Dynamic Range) and Sample Rate (Frequency Range)

Resolution (Dynamic Range)

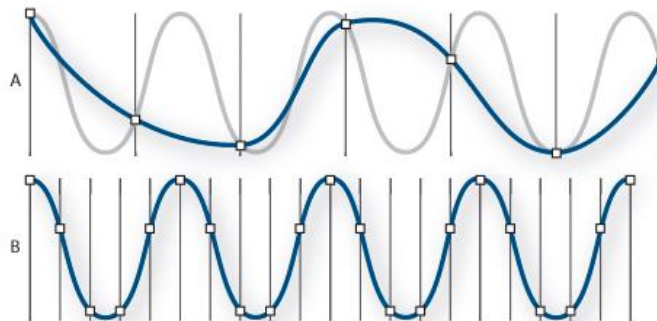
The resolution of the ADC is determined by the number of bits that are store in the result of the conversion.

For example, if our ADC is an 8-bit type then we can have 2^8 or 256 possible values for or converted analog voltage. If the input range of our ADC is 0-3.3V then this mean the smallest voltage we can discern is $3.3V/255$ or 12.9mV. If our ADC is a 16-bit type, then we can have 2^{16} or 65,536 possible values for or converted analog voltage. If the input range of our ADC is 0-3.3V then this mean the smallest voltage we can discern is $3.3V/65,535$ or 50.3 μ V.

Resolution equates to dynamic range of the signal. You can think of this by thinking of a loud and quite signal in succession. If we want to detect the small signal the loud signal will clip the inputs to our ADC and will not be detected, however, if we want to detect the loud signal and we adjust the input gain to the ADC then the small signal may be so low that we cannot detect it within a single bit of resolution.

Sample Rate (Frequency Range)

The sample rate of the ADC is determined by the clock rate at which the analog signal is converted to a digital value. The faster the rate of conversion the higher of frequency a signal can be detected. The highest signal that can be faithfully reproduced is known as the Nyquist Frequency and is $\frac{1}{2}$ the sample rate. When the input frequency is greater than the sample rate aliasing occurs and the signal captured appears lower in frequency than than the actual signal desired.



Calculating Sample Values

When working with your ADC you are going to need to determine the value that your ADC will return when a specific voltage is applied to the input. To calculate this, you can use the following equation:

$$ADC_{reg} = \frac{V_{in}}{V_{ref}} (2^{bits} - 1)$$

Where ADC_{reg} is the result of the conversion, V_{in} is the input voltage to the ADC, V_{ref} is the reference voltage (usually the maximum voltage that ADC will sample) and bits are the number of bits of resolution for the ADC we are using.

Sometimes you need to work backwards, that is we have the value from in the ADCreg but want to know what voltage on the input gave us this value. In this case we solve the above equation for Vin and get:

$$V_{in} = \frac{ADC_{reg}}{(2^{bits} - 1)} V_{ref}$$

Example

We have an input voltage of 1.5V a reference voltage of 3.3V and the resolution of our ADC is 10 bits.

$$ADC_{reg} = \frac{V_{in}}{V_{ref}} (2^{bits} - 1) = \frac{1.5V}{3.3V} (2^{10} - 1) = 465_{10} = 1D1_{16}$$

This will give us 465 or 0x01D1. We can verify this by using the equation that is solved for Vin.

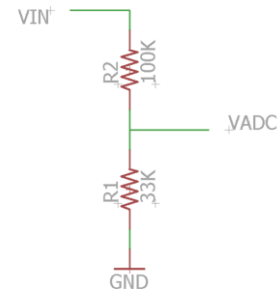
$$V_{in} = \frac{ADC_{reg}}{(2^{bits} - 1)} V_{ref} = \frac{465_{10}}{(2^{10} - 1)} 3.3V = 1.5V$$

Measuring Voltages Out of Range

Most of the time ADC pins on the PIC are operated between 0V and Vcc of the chip. There are exceptions that can be read about in the ADC section of the datasheet. If Vcc of our chip is 3.3V but you want to measure a signal outside of this range how can this be accomplished?

Attenuation

If your signal is between 0V and some larger voltage then the signal need only be attenuated with a voltage divider. I like to choose resistor values such a 33K and 100K. This make the maximum voltage that can be measure 13.3V (33K/100K) = 3.3V when 13.3V is applied to VIN the VADC will see 3.3V. Of course, it would be a good idea to make sure that your maximum voltage is not exactly 13.3V, for if it were and your resistors are not perfect then you will not get exactly 3.3V on VADC. If VADC is too large you could damage the microcontroller.



Amplification

If your signal is small (less than the resolution of the ADC or you would like greater dynamic range on a small signal) or if your signal contains a negative voltage then an amplifier will be needed to condition the signal.

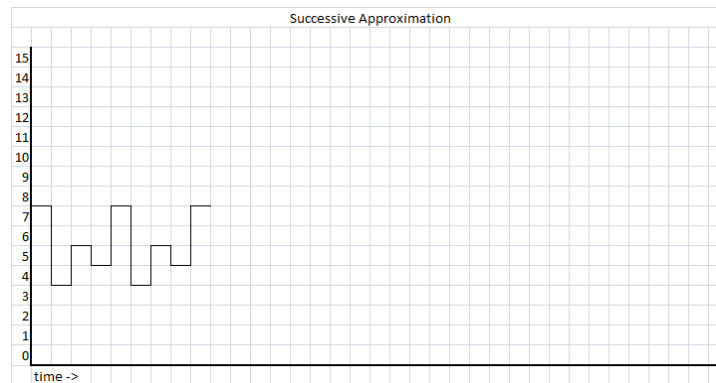
Discussion of amplifier circuits is beyond the scope of the class, but MtSAC offers a fine class on the subject called electronic devices.

The PIC ADC

Successive Approximation

A problem arises with using a stair step, especially when a large number of bits are used. If your ADC is 16-bits, there the ramp would have to step through 2^{16} or 65536 levels with each pass trying to detect the unknown voltage. If you're trying to convert a signal in a hurry it will take some time to detect your signal. One possible way to speed it up is if you restart your ramp at zero as soon as you find your unknown voltage, but this has problems as well. This will make the time between samples depend on your unknown voltage.

A common solution to this issue is known as successive approximation. Successive approximation alters a single bit at a time in a binary search to hone in on the unknown voltage. The unknown voltage can be discerned in the same number of steps as bits of conversion.



Analog Multiplexer

In addition to using successive approximation, the PIC contains an analog multiplexer that allows you to hook up several analog signals to a single ADC. Details on the operation of the ADC are included in the PIC datasheet in the ADC section.

Integer Division (The modulus (%) and division operators (/))

To discuss division a quick reminder on terminology used when talking about the values use when doing a division operation.

$$\text{quotient} = \frac{\text{dividend}}{\text{divisor}} = \frac{17}{5} = 3 \text{ r } 2$$

When operating on integer values in C and performing division the results of the division is truncated is the number we are trying to divide is not evenly divisible. For example: 17 divided by 5 is 3 with a remainder of 2. In C we can get quotient by using the division operator / and we get the remainder by using the modulus operator %.

```
uint8_t dividend = 17;
uint8_t divisor = 5;
uint8_t quotient;
```

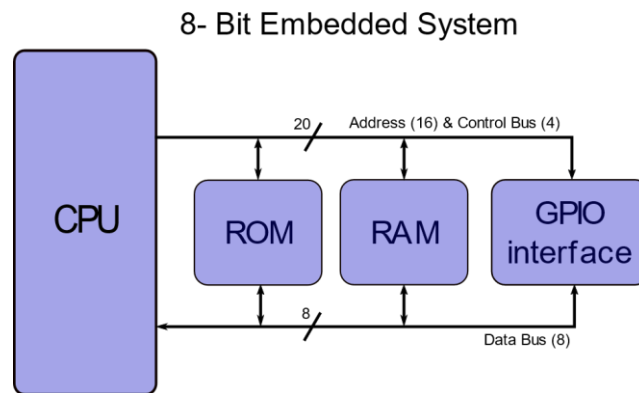
```
uint8_t remainder;  
  
quotient = dividend / divisor; // quotient = 3  
remainder = dividend % divisor; // remainder = 2
```

Week 8 – Mid Term / Lab Makeup

Week 9 – SPI and Digital to Analog Converter (DAC)

Chip to Chip Communications

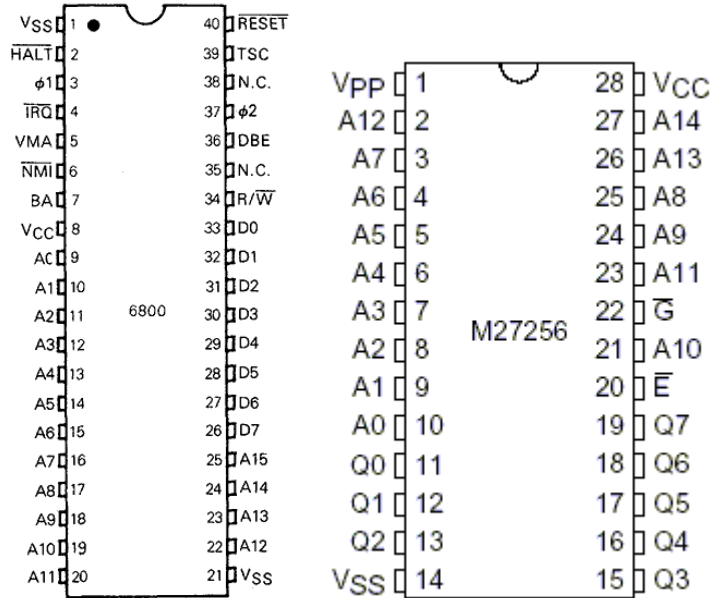
If you think back to the beginning of this class you will remember that the microcontroller is a microprocessor with memory, I/O and other peripherals that have all been put onto a single chip. But before the advent of the microcontroller the memory, I/O and peripherals were all wired to the microprocessor to give the functionality that we enjoy in a single chip today. A simplified block diagram of such a system would look like this:



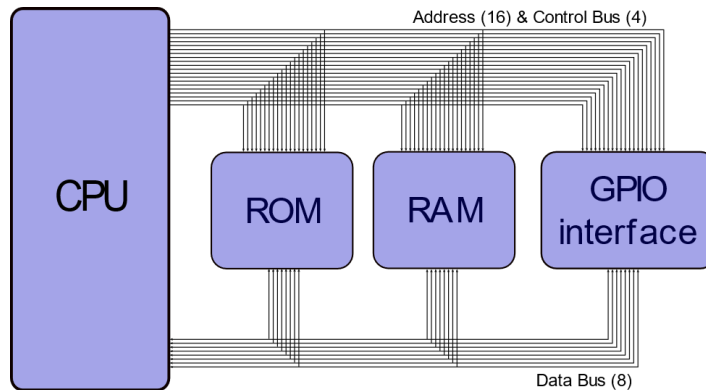
This diagram shows the connections between the CPU, ROM, RAM and GPIO using buss notation. A bus is a group of connections that are drawn in a schematic as a single line with a slash (/) and a number next to the slash showing number of wires in the bus. The diagram is further simplified in that there is external (to the CPU and memory) chip select logic that is not show that is needed to determine which chips are being selected by the CPU in any given cycle of operation.

Buses

To understand how buses work you need to look at the pinout of chips and how they connect together. Below is shown a Motorola 6800 Microprocessors (not microcontroller) and a 27256 EPROM chip. The address lines of each chip are labeled Ax (where A is short for address and the x is a bit number). When a buss is drawn all the line for a given bus move on a single line. This makes drawing and reading schematics easier.



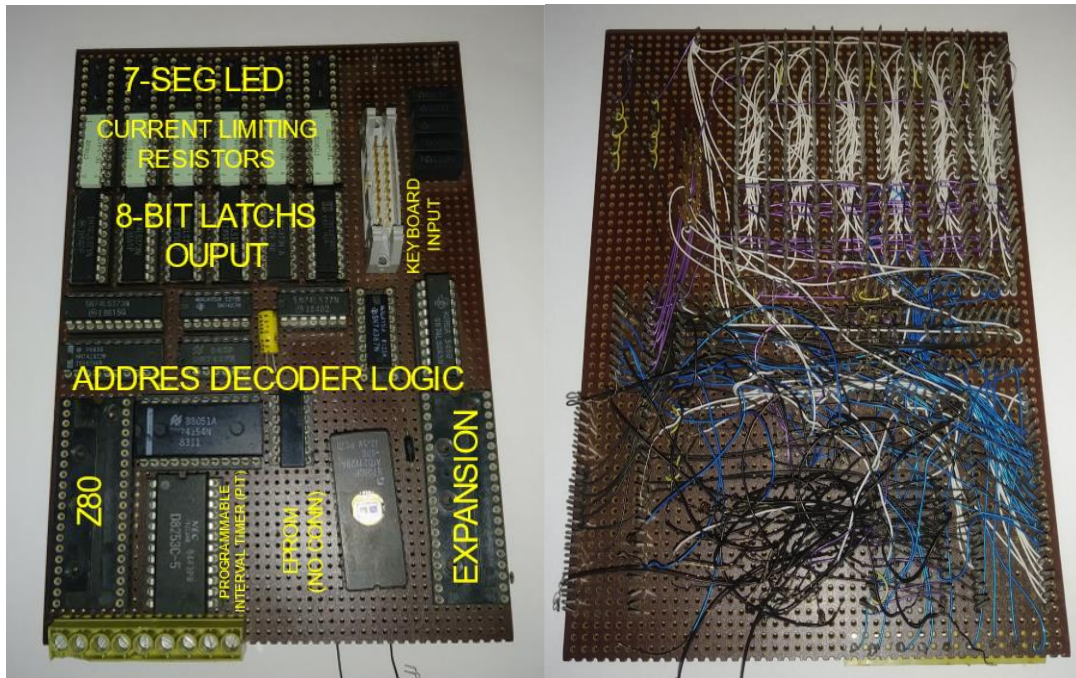
If the buses were not drawn the diagram would look more like this:



Each of the signal wires shown in the above diagram is a binary logic line (0 or 1) or a single bit per line. These pictures help illustrate that why an 8-bit computer is called 8-bit, namely that there is 8-bits in the databus. In an 8-bit computer there are typically 16 address lines. The address bus is an output from the CPU that is used to select a single memory location (or GPIO port). There are also some control signals that orastrate communications between the CPU and external devices. These control signal provide timing (E) determine if the CPU is reading or writing (R/W). The 8-bit computer system illustrated above is far simpler than the single PIC32 microcontroller used in this class.

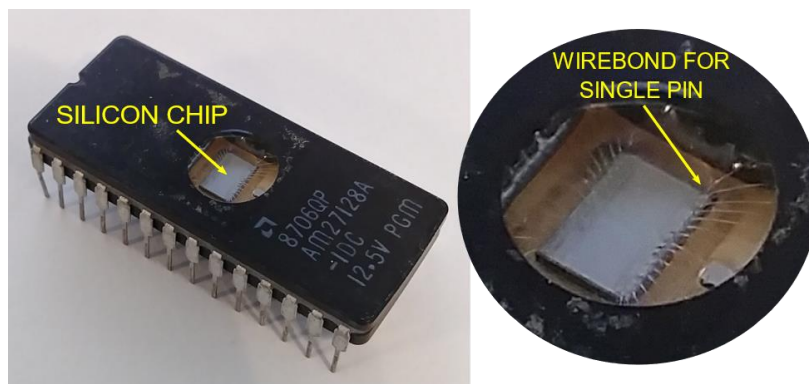
The Problem: Too Much Wire, Large Parts

So why look at this system? It is to help illustrate a point that when you build a system like this it requires lots of interconnections. Each interconnect is a wire. If you were to prototype a system like this you would have place a physical wire for each connection in the system. Below is an example of a project that I built when I was a student at MtSAC using the Z80 microprocessor. At this time microcontrollers were just coming on to the scene and they were not yet being taught at MtSAC. You can see in this system that there are lots of wires that are used to connect memory to the microprocessor (which would go in the 40 pin 600 mil spacing socket on the left if it wre installed).



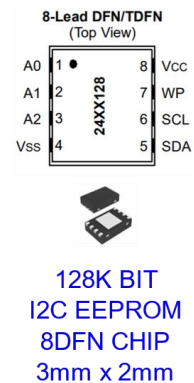
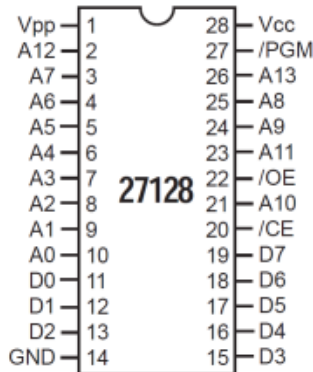
Jacob Christ student project: Component Side (left) / Wirewrap Side (right)

Early designers of microcontroller system quickly realized all these interconnections meant that chips needed to be physically large to accommodate all these connections. As you can see in the picture of the 28 pin EPROM IC below, the actual silicon chip is quite small compared to the physical size of the housing that is required to break out all of the pins that need to go to the chip.



The Solution(s)

Some of the solutions offered are a reduction of the number of wires needed when communicating from one chip to another. This is accomplished by changing the architecture of the system from using parallel wires on a bus to sending data on serial connections from one chip to another. As we have seen we can easily achieve bi-directional communications by using two wires and a ground connection using an UART. UARTS are quite popular for use in system to system communications, but when we are doing chip to chip communications there are some alternatives that are more common. The two most common solutions are known as SPI (Serial Peripheral Interface) and I2C (Inter-Integrated Circuit).



The Tradeoff's

The primary difference between UART and SPI or I2C is that they are synchronous which means that they have a clock line synchronizing the data being transmitted or received. Additionally, SPI has a physical chip select pin for each device we want to communicate with. The chip select is a signal that originates at the microcontroller to select the chip. I2C uses a virtual chip select by sending data on indicating which chip is intended to be communicated with.

SPI, and technologies like SPI such as I2C, have allowed chips to get smaller (by reduction of pin count due to serial nature rather than parallel) and in turn allowed PCB's to get smaller since less traces are required on the board for chip to chip communications. Most mid-range and high-end PICs have the capability to communicate with other ICs and devices.

Feature	I ² C	SPI	UART
I/O Pins	2	4	2
Top Speed	400kHz	20-40 MHz	~1 MHz
Address Mode	Protocol Overhead	External Chip Select	None
Master Mode	Multi Master	Single Master	Depends on Protocol
Other Issues	Since the chip address is defined in the protocol this may limit the number of a particular chip type that can be connected to the bus	Implementations vary From vendor to vendor and can cause confusion when setting up MSSP	UART's require crystal oscillators which make the system and the chip more complicated. A common place where UART would be used is to talk to a GPS receiver that has an on-board microcontroller and

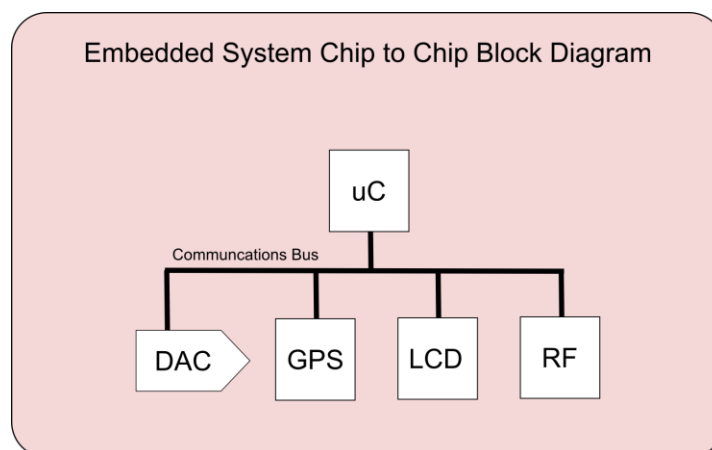
Expanding the Functionality of the Microcontroller

Although the PIC microcontroller is an amazing device with multitudes of functionality sometimes the functionality that one might want in an embedded system is not available in a single chip. Additional functionality can be added by adding additional chips to our system that extend the capabilities of the microcontroller and there are thousands of chips that can be easily interfaced to microcontrollers. Without thinking too hard here is a small list of cool things that the PIC cannot do: There is no built-in display, GPS, DAC, it only has two USARTs, no ability to do RF communications, mass storage in the form of EEPROM or flash memory. Below is a partial list of some devices available to extend the functionality of a microcontroller.

Device Name	Manufacturer	Example Model Number	Interface
MP3 Decoder	ST Microelectronic	STA013, STA015, etc...	SPI, I2C
Ogg Vorbis / MP3 / AAC / WMA / MIDI audio codec	VLSI Technology	VS1053, etc..	SPI
MMC/SD Card	To many to name	To many to name	SPI, 4-bit native
Pressure Sensor	National Semi	LM74	SPI
Real Time Clock	Dallas	DS1305, DS1306...	SPI
Temperature	Analog Device Inc.	AD7816, AD7817, AD7818	SPI
Camera lenses	Canon	Canon EF lens mount	SPI
EEPROM	Microchip	25LC1024	SPI
LDC Drivers	Sharp + others		SPI
GPS Receiver	Delorum		SPI, USART

If there is not a chip available off the shelf to do what you are looking for, this does not mean that it cannot be done, but this discussion is beyond the scope of this class.

To gain the functionality our firmware will need to be able to send and retrieve information from these devices. Shown below is a block diagram of how our system might look with several chips connected to the PIC.



In this section we will study chip to chip communications with the PIC32 with emphasis using SPI communications module. But before we get started we will take a quick look at I2C communications.

I²C

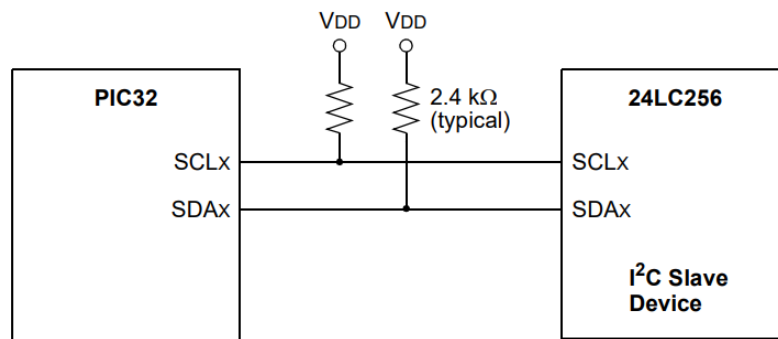
I²C (Annunciated eye-squared-see or eye-two-see) is an acronym for Inter-Integrated Circuit. The disadvantage is that since it does not have chip selects lines that chip address need to be transmitted over the serial line. This adds overhead to the already slower than SPI communications, increases the complexity of the communications protocol and adds complications as to possibly having chips with overlapping addresses (since the address are built into the chip from the manufacture). Despite the disadvantages of I²C the advantages are tremendous. The major benefit of I²C is that it requires only two I/O lines for all devices.

The choice to study only SPI in this class should not provide a bias as to which to use in a project since both have their place. The choice is a matter of practicality in a time limited form. At some future point a module may be developed for self study for interfacing an I²C to a PIC and it is highly encouraged that you pursue a self study of the technology after the class has completed.

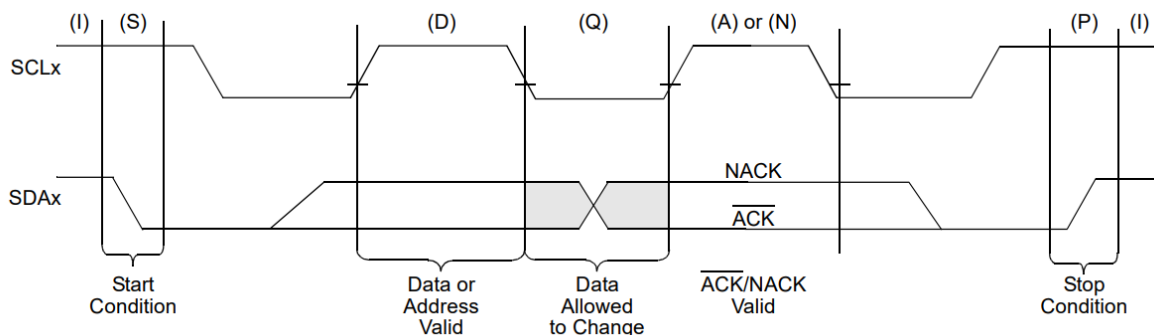
Signals (two)

SCL (Serial Clock) – Output from the master

SDA (Serial Data) – Bidirectional communications



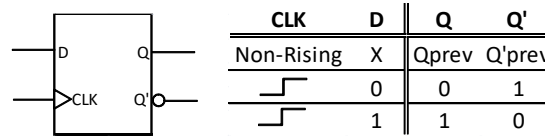
Typical I²C Interconnection Block Diagram



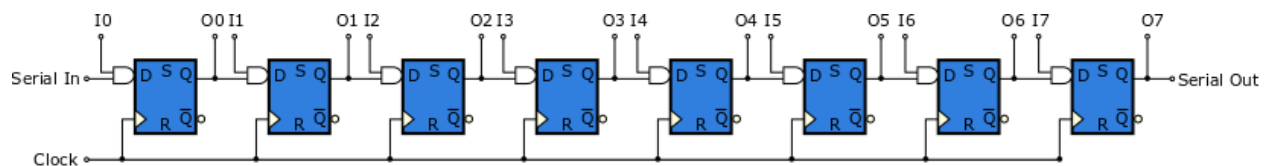
I²C Bus Protocol State Diagram

SPI

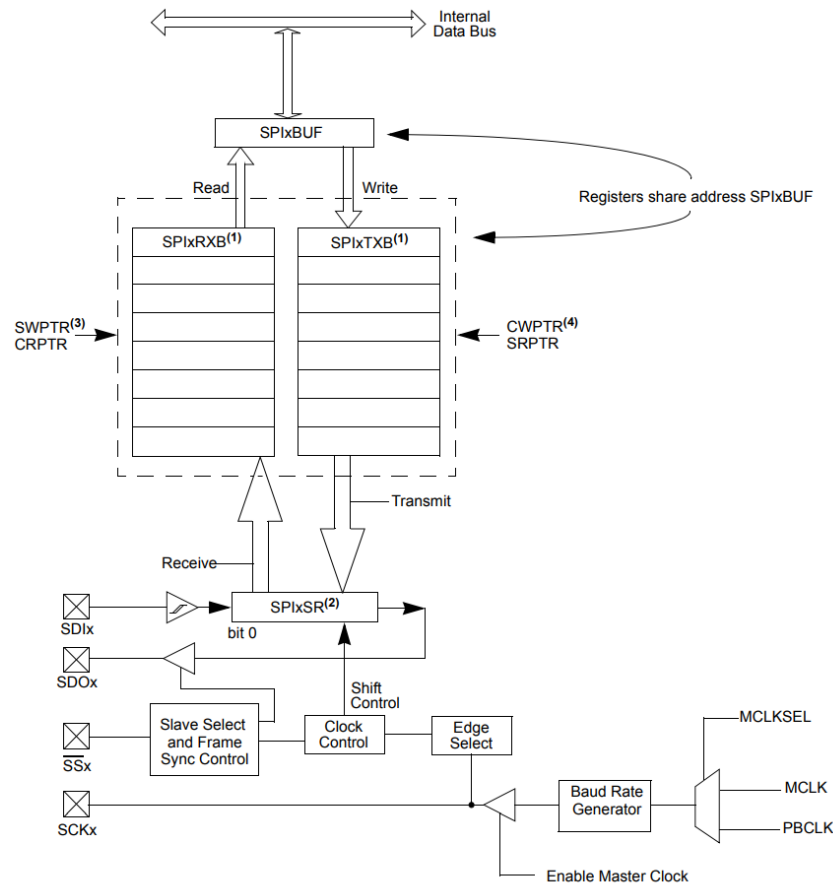
SPI (announced es-pee-eye) is an acronym for Serial Peripheral Interface. To understand SPI hardware we should first review parallel to serial and serial to parallel shift registers. But first a quick reminder on how D-Flip Flops work:



The circuit below is a very simplified diagram that can act both as a parallel to serial shift register or a serial to parallel shift register. For parallel to serial operation first the S inputs must be pulsed to make all Q outputs logic 1. Next the Serial In is set to logic 1. Then with a single clock all logic inputs I0-I7 can be loaded with a single clock pulse. At this point Serial out is presenting the MSB from the shift register. To shift this data out, the inputs I0-I7 are then brought to a logic level 1 to allow data to shift through the register. The clock is pulsed seven more times, with each pulse Serial out is presented with the next most significant bit. As the data is clocked out, new serial data is clocked in.



The above is the basis for the SPI system on the PIC32. The actual hardware on the PIC32 is quite a bit more complicated and includes additional buffers for both data shifting into and out of the PIC32. The additional buffers allow for automated shifting of data without bogging down your program. The way the buffers work is that you load your data into the buffer then your program continues while the hardware in the PIC 32 shifts the data out for you. The PIC32 libraries for the that are part of the chipKIT project do not take full advantage of this hardware. Hint, if your ambitious you could contribute your time to the chipKIT project to improve these libraries. Indeed many companies look to contributions to open source projects when evaluating candidates for employment.



PIC32 SPI Block Diagram

Signals (three + one chip select for each device)

As you can see from the above block diagram for each SPI module in a PIC32 there are four processor pins associated with it. These control signals are described below:

- SCLK — Serial Clock, output from master (shared with all chips)
- SDO (Serial Data Out) or MOSI (Master Output, Slave Input)—Shared with all chips
- SDI (Serial Data In) or MISO (Master Input, Slave Output)— Shared with all chips
- SS' (Slave Select) — Input when configured as a slave device. Can be used a CS' (chip select when master)

In addition to the four dedicated SPI signal per SPI module on a PIC32 we will need to assign one GPIO pin per device we need to communicate with.

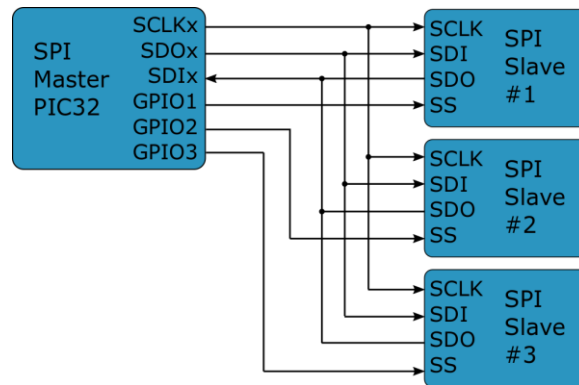
CS' (Chip Select)

The chip select line for a SPI device is typically active low (which if you recall from digital means that it is logic low when the signal is asserted and logic high when it is not asserted).

Master and Slave

SPI minimally needs one master and one slave device. The PIC32 can be either a master or slave device, but we will study only the PIC32 as a master.

The SPI master is the device that controls the communications and generates the clock signal. There can be only one master on an SPI bus. The slave device provides additional functionality to the master and there can be many slave devices on a single SPI bus. A typical SPI system block diagram might look something like this where there is a single master and many slave devices:



If we contrast SPI to I2C or UART communication we can see that the disadvantage of SPI is that it requires a minimum of three I/O lines for communications (four if bidirectional communications is required) and an additional chip select line for each additional chip added to the system. So, if four chips are to be interfaced to the microcontroller and bi-directional communications is required this will consume seven I/O lines of your chip (SCLK, SDO, SDI + 4 Chip Selects). A similar system implemented in I2C or with a UART would require only two I/O lines. The required chip select line makes the protocol easy to understand and interface to simple devices such as shift registers, it is also the fastest of the three serial communication protocols we have looked at. These are why we are studying it.

Clock Frequency

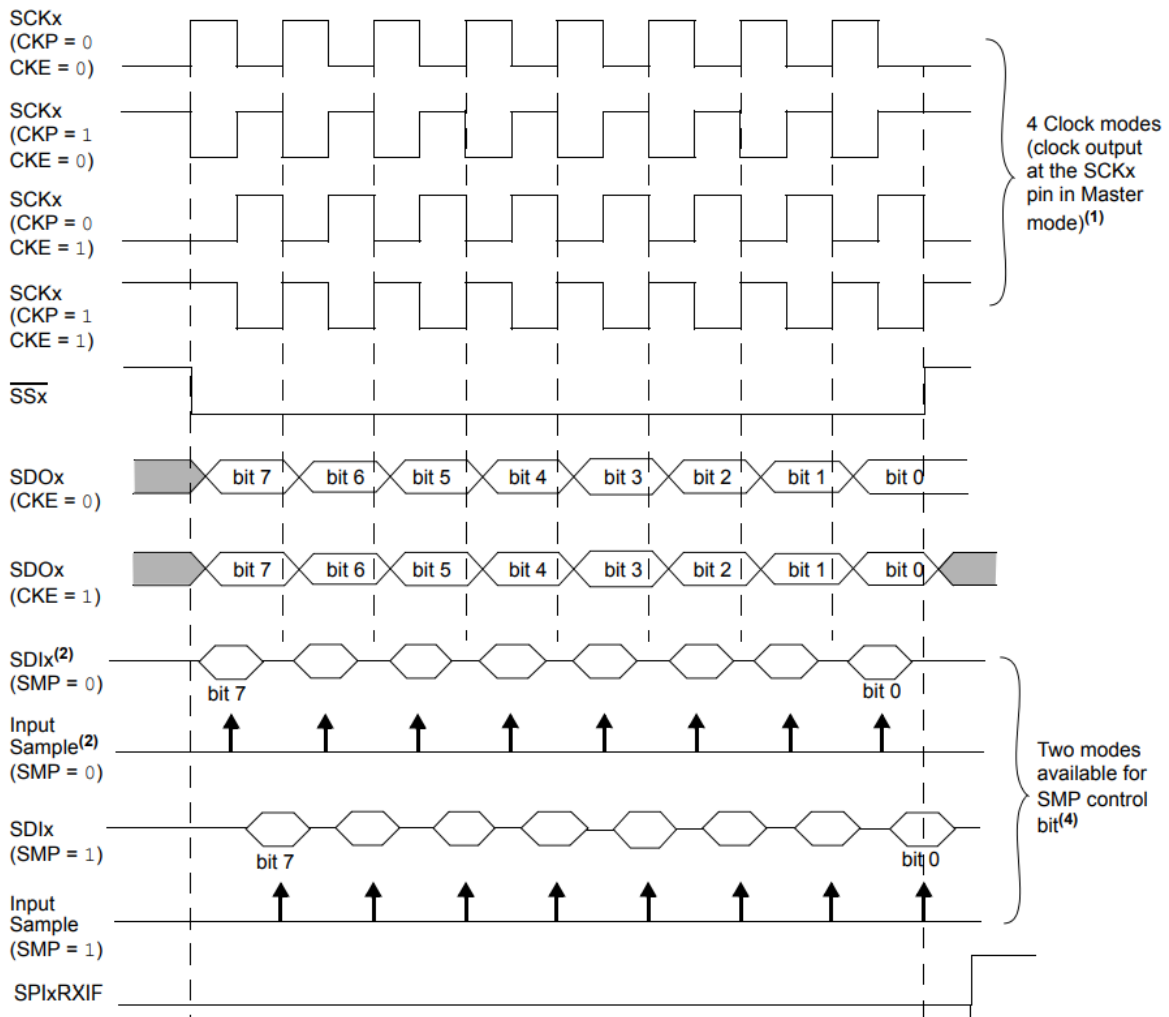
The frequency that the clock runs out is configured by specifying dividing frequency from the fundamental operating frequency of the PIC32. The clock dividers are available in powers of two due using flip-flop frequency dividers to accomplish this task. The reason we use a divider is because the hardware does not know what frequency it is running at. So, we simply specify a divisor value. Can you think of some advantage and disadvantages to this approach? How could we specify frequency if we only had hardware in the chip that was capable of frequency division?

Clock Phase and Polarity

There is one final but especially important consideration when using SPI. Setting clock phase and polarity. There are four modes available.

Mode	Clock Polarity (CPOL/CKP)	Clock Phase (CPHA/CKE)
SPI_MODE0	0	0
SPI_MODE1	0	1
SPI_MODE2	1	0
SPI_MODE3	1	1

The most common is SPI_MODE0 but you need to compare the timing diagrams of the slave device to the timing diagram from the PIC32 datasheet to know the correct SPI mode to use.



PIC32 SPI Timing Diagram Showing SPI Clock Phase and Polarities

Configuring SPI

Using the wiring to configure the SPI is as simple as doing the following.

1. Include SPI.h at the top of your sketch.

```
#include <SPI.h>
```

2. In setup() function, add SPI.begin()

```
SPI.begin();
```

3. Set the clock divider to one of the following values (80MHz clock divided by...):

SPI_CLOCK_DIV2
SPI_CLOCK_DIV4
SPI_CLOCK_DIV8
SPI_CLOCK_DIV16
SPI_CLOCK_DIV32
SPI_CLOCK_DIV64
SPI_CLOCK_DIV128

```
SPI.setClockDivider(SPI_CLOCK_DIV64); // Low frequency due to bread boarding
```

4. Set the SPI mode, clock polarity and phase:

Mode	Clock Polarity (CPOL)	Clock Phase (CPHA)
SPI_MODE0	0	0
SPI_MODE1	0	1
SPI_MODE2	1	0
SPI_MODE3	1	1

```
SPI.setDataMode(SPI_MODE0);
```

5. Set the chip select pins you wish to use to output where chip_select_pin_number is the number of the pin of the board that will be used for the chip you are connecting the chip select to.

```
pinMode(chip_select_pin_number, OUTPUT);
```

The Arduino SPI API

The Arduino SPI API is documented here:

<https://www.arduino.cc/en/Reference/SPI>

Digital to Analog Converter (DAC)

A DAC is just what it sounds like and provides the inverse functionality of an ADC. That is to say it takes a digital value as input and outputs an analog value (usually voltage or current) that can be used to control an analog circuit.

The same rules for resolution and sample rate apply for a DAC as for an ADC and the calculations for determining work in reverse. The equations for calculating the output voltage is shown here.

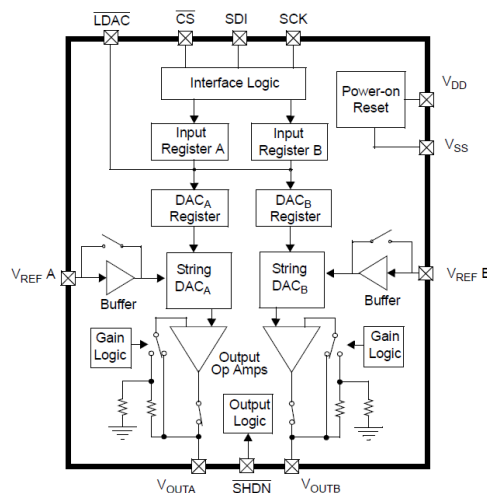
$$V_{out} = \frac{DAC_{reg}}{(2^{bits} - 1)} V_{ref}$$

And the equation form above solved for the DAC register value is shown here.

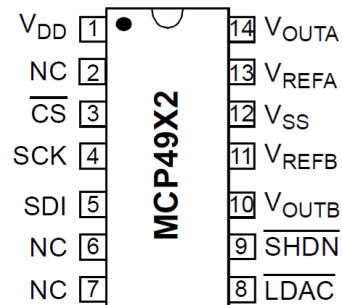
$$DAC_{reg} = \frac{V_{out}}{V_{ref}} (2^{bits} - 1)$$

MCP4902

The DAC we will be studying is the MCP4902 from Microchip. This DAC has 8-bits of resolution and two channels in a single chip. This chip is part of a family and there are 10-bit and 12-bit variations available. Once you know how to work with the 8-bit chip it will not be difficult to extend your code to communicate with one of the other chips. A block diagram and pin out from the datasheet for the chip is shown below.



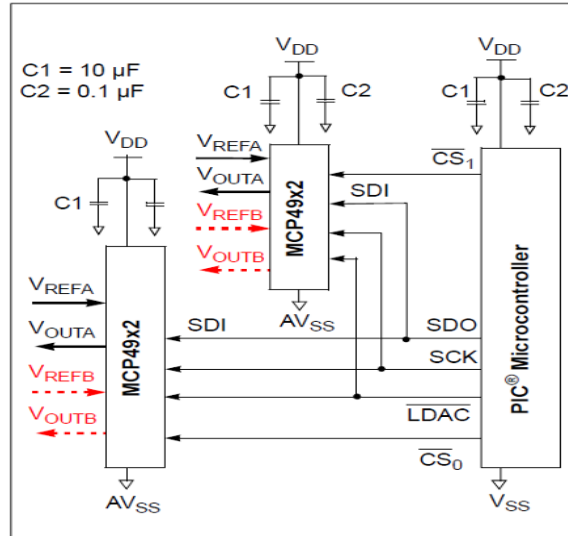
14-Pin PDIP, SOIC, TSSOP



MCP4902: 8-bit dual DAC
MCP4912: 10-bit dual DAC
MCP4922: 12-bit dual DAC

Typical Connection

The PIC does not have a built in DAC, so we will use an external DAC connected to the SPI bus of the MSSP module. Since the DAC takes digital data as input and outputs an analog voltage the MISO line of the SPI bus is not needed and all communications can take place using only the MOSI, SCLK and CS lines. Show below is a typical connection diagram for connecting an SPI device to a PIC microcontroller.



(form MCP4902 Datasheet)

Communications Timing Diagram

When interfacing to a new SPI device it is important to understand the timing diagram for the device. This allows us to understand what data needs to be sent or received from the chip to establish communications with the chip. The diagram below is for the MCP4902 8-Bit DAC. The signals shown in the diagram are CS, SCK, SDI, LDAC and Vout. All of these signals except Vout are outputs from the PIC and must be assigned to pins on the chip. The Vout is the voltage out of the DAC. The LDAC is not part of the SPI bus and is specifically for this DAC chip. The LDAC is an active low signal that allows synchronizing the two channels of the DAC by bringing the line low after both DAC registers have been loaded.

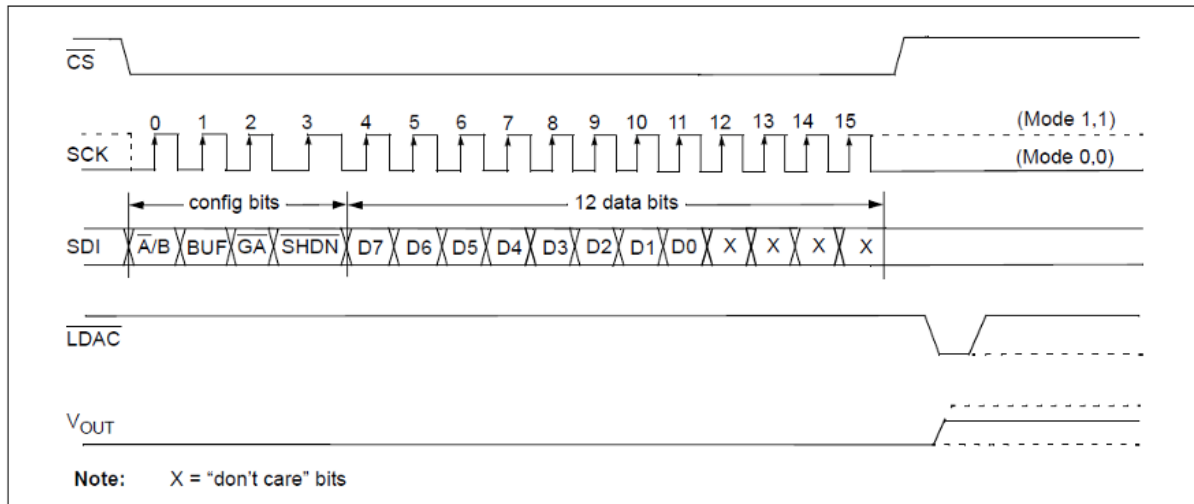


FIGURE 5-3: Write Command for MCP4902 (8-bit DAC).

The interesting line to look at in this timing diagram is the SDI line. If you look closely what you see is that each bit that is transmitted from the PC has a name, and of course a function. The bits are described in detail in the MCP4902 datasheet and the summary data from the datasheet is reproduced here:

Note that the x bits are unknown or don't care bits and are not needed to use the chip.

Week 0x0A – Object Oriented Programming (C++ Classes and Objects)

An introductory look at classes and object-oriented programming as an organization tool.

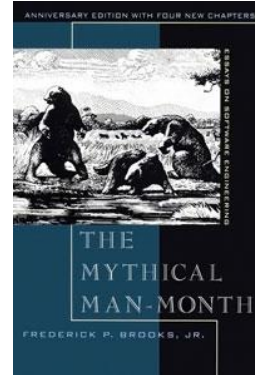
The Problem: Brooks Law

As we have learned in this class, software can be complicated to create.

In 1975 the book “The Mythical Man-Month” by Fredrick P. Brooks was published. Brooks' observations are based on his experiences at IBM while managing the development of OS/360. He had added more programmers to a project falling behind schedule, a decision that he would later conclude had, counter-intuitively, delayed the project even further. -Wikipedia

(https://en.wikipedia.org/wiki/The_Mythical_Man-Month)

The primary cause of this problem arises from how intercommunications between group members grows exponentially as the number of group members grows linearly. The number of communications paths in a group is given by the following formula:



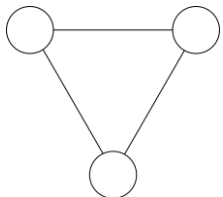
Group intercommunication formula: $p = n(n - 1) / 2$

Where n is the number of group members and p is maximum the number of communications paths between any two members. This is illustrated below where a circle represents a group member and a line between two circles represents a communication path between any two members.

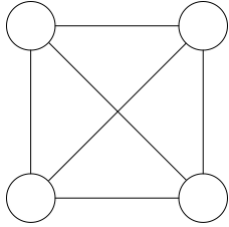
Example of $n = 2$, $p = 2(2-1)/2 = 1$:



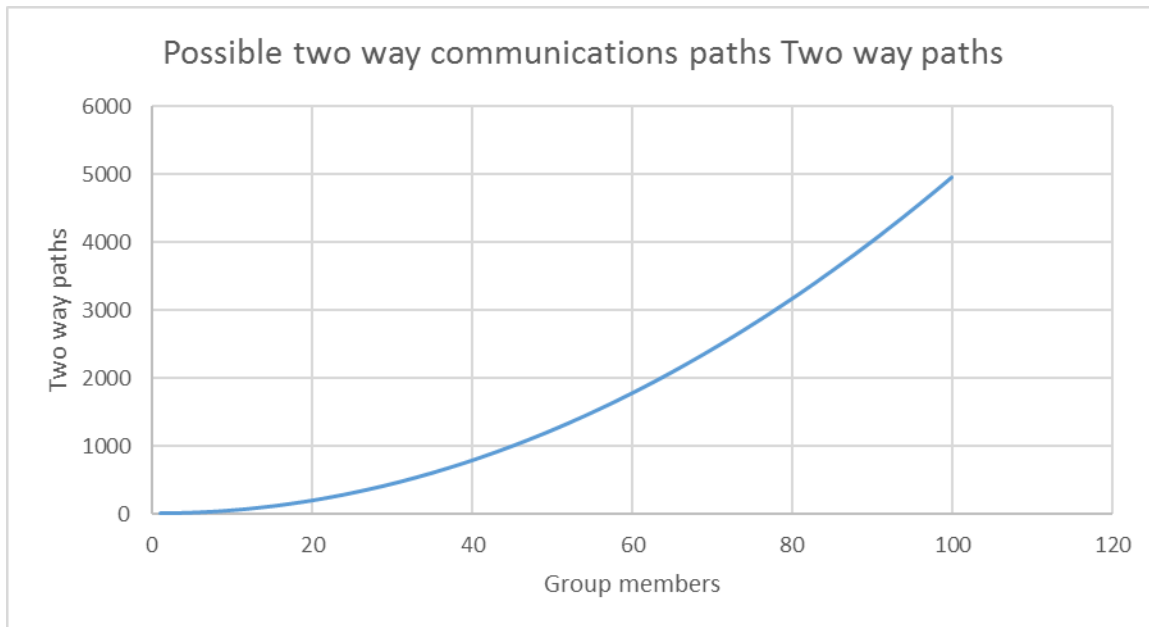
Example of $n = 3$, $p = 3(3-1)/2 = 3$:



Example of $n = 4$, $p = 4(4-1)/2 = 6$:



Example of $n = 100$, $p = 100 \cdot (100 - 1) / 2 = 5000$:



One of Many Solutions (Segmentation)

There are many tools to help resolve the problem presented by Brooks' Law but in depth study of possible solutions is beyond the scope of this class. If you are interested in this topic or find yourself on a team facing such problems this would be a good topic of independent study.

The solution we are going to look at in this class that relates specifically to this issue is called segmentation.

Simply put, segmentation helps by minimizing the communication overhead between team members. A problem is divided up into smaller sub-problems. Each sub-problem is solved by a smaller team. A top-level team is responsible for sub-system integration into the larger problem trying to be solved.

So you want to build something really ambitious?

Start with what you want

Right now you have nothing but an idea. Write down what the manifestation of your idea will be when you complete your design and build process.

Divide design into sub-problems

Small Teams

Remember, the more people on a team the more communications path that are needed and the slower the project will move.

Timeline

Each sub-problem should have similar timelines. If many teams are to be used in parallel this helps to get all the pieces to come together at the same time.

As a rule of thumb, if a specific task is estimated to take longer than a day, then it is probably not defined with enough granularity.

The Interface

The interface to the sub-problem should be well defined (not a moving target). This assures that the parts of the project that need to connect to this system need not have to change much.

Repeat

Do this over and over until all sub-problems are sufficiently compartmentalized for the size of your team.

Example

1. Self-driving car
 - 1.1. Mechanics
 - 1.1.1. Chassis
 - 1.1.2. Motor
 - 1.1.3. Drive Train
 - 1.1.4. Fairing
 - 1.1.5. Breaking
 - 1.1.6. Wheels
 - 1.2. Electronics
 - 1.2.1. Motor Ignition Control
 - 1.2.2. Security
 - 1.2.3. Entertainment
 - 1.2.4. Windows
 - 1.2.5. Safety (Headlights / Break Lights)
 - 1.2.6. Driver Console
 - 1.2.7. Cameras
 - 1.2.8. LIDAR
 - 1.2.9. GPS
 - 1.3. Software
 - 1.3.1. Driver Console
 - 1.3.2. Cruise Control
 - 1.3.3. Vision Software

- 1.3.4. Car Detection
- 1.3.5. Human Detection
- 1.3.6. Road Detection
- 1.3.7. Obstacle avoidance
- 1.4. Legal Team
 - 1.4.1. Compliance with existing vehicle design laws
 - 1.4.2. Exploration of new legal ramifications of new technologies
- 1.5. Marketing

Examples of projects that could benefit from this approach

Space ship

UAV Drone

3D Printer / CNC Machine

Purpose of OO Programming

Object oriented programming is software design philosophy that of implementing segmented design in software. In C++ a structure called a class is provided to help implement segmentation.

Classes

A class defines a set of attributes (variables) that can be affected by methods (functions) all contained within the namespace of the class.

Objects

Classes are just definitions. The realization of a class in a program is called an object.

Organization Tool

Using objects in your program enforces a level of organizational discipline. It is easy to bypass this discipline but doing so negates the use in the first place.

Code Reuse (Nut and Bolt Model of Programming)

You don't want to redesign the wheel, or do you?

A word of caution

Just because you use object-oriented paradigms such as objects and classes, doesn't mean they are being used correctly. A class can be poorly designed and make programming more difficult.

Class as an Abstract Data Types

Up until this point any variable that we have declared has been what is known as a "primitive type". Typically, in the field of computer science, something referred to as a primitive implies that it can get no simpler. If we were thinking about primitive shapes a list might include things like "circle, triangle, square". From primitives we can build more complex things, in our case we are going to build what are known as "abstract data types" or simply an ADT. The mechanism in the C++ programming language to

build an ADT is what is known as a class. A class is a collection of primitives and other structures that are in a common collection to create something altogether new. The whole thing is really quite simple.

Class Definition and Abstract Data Types

Up until this point any variable that we have declared has been what is known as a "primitive type". Typically, in the field of computer science, something referred to as a primitive implies that it can get no simpler. If we were thinking about primitive shapes a list might include things like "circle, triangle, square". From primitives we can build more complex things, in our case we are going to build what are known as "abstract data types" or simply an ADT. The mechanism in the C programming language to build an ADT is what is known as a structure. A structure is a collection of primitives and other structures that are in a common collection to create something altogether new. The whole thing is really quite simple. And a structure might look like this:

C++ Class Template

The following is a subset of functionality

```
class ClassName
{
    private:
        type private_variable;
        type private_method1 (type parameter)
        {
            // Code
            private_variable = new_value;
        }
    public:
        type public_variable;
        // Constructor method (function)
        ClassName(type parameter)
        {
            // Constructor Code
            public_variable = new_value;
            private_variable = new_value;
            public_method1(parameter);
            private_method1(parameter);
        }
        // Public Function
        type public_method1(type parameter)
        {
            // Code
            private_method1(parameter); // Okay
            private_variable = new_value; // Okay
        }
};
```

Qualifiers

Public

Member variable and functions can be accessed by any users.

Private

Can be accessed by class methods (functions) only.

Usage

```
void setup()
{
    ClassName object(100);
    object.public_variable = new_value;
    object.public_method1;

    object.private_method1; // CANNOT DO THIS
    object.private_variable = new_value;
}
```

Functions outside of in class structure

Member functions (methods) can be defined outside of the class structure. When doing so the function definitions for the above class would look something like this:

```
type ClassName::private_method1 (type parameter)
{
    // Code
    private_variable = new_value;
}

ClassName::ClassName(type parameter)
{
    // Constructor Code
    public_variable = new_value;
    private_variable = new_value;
    public_method1(parameter);
    private_method1(parameter);
}

type ClassName::public_method1(type parameter)
{
    // Code
    private_method1(parameter); // Okay
    private_variable = new_value; // Okay
}
```

Constructors

There is a special member function that must be public called the constructor. The constructor has the same name as the class and is called automatically when the class is instantiated.

A (not so) practical example

```
// Arduino IDE 1.6.7, chipKIT core 1.1.0

class NPC {
  private:
  public:
    uint8_t x;
    uint8_t y;

    NPC();
    void move();
};

class BOARD {
  private:
  public:
    static void draw(NPC *npc, uint8_t max_player);
};

const uint8_t max_x = 30;
const uint8_t max_y = 20;

NPC::NPC() {
  x = random(0, max_x);
  y = random(0, max_y);

  Serial.print("x=");
  Serial.print(x, DEC);
  Serial.print(" y=");
  Serial.print(y, DEC);
  Serial.println("");
  delay(1000);
}

void NPC::move()
{
  uint8_t d;
  d = random(0, 3);
  if( d == 0 ) {
    x++;
  }
  else if( d == 1 ) {
    x--;
  }
}
```

```
    }

    d = random(0, 3);
    if( d == 0 ) {
        y++;
    }
    else if( d == 1 ) {
        y--;
    }
    x++;
    y++;
    if( x >= max_x ) x = 0;
    if( y >= max_y ) y = 0;
    if( x < 0 ) x = max_x - 1;
    if( y < 0 ) y = max_y - 1;
}

void BOARD::draw(NPC *npc, uint8_t max_player)
{
    uint8_t x;
    uint8_t y;
    uint8_t player;
    uint8_t draw;

    y = 0;
    while( y < max_y ) {
        x = 0;
        while( x < max_x ) {
            draw = 0;
            player = 0;
            while( player < max_player ) {
                if( x == npc[player].x && y == npc[player].y )
                    draw++;
                player++;
            }
            if( draw > 0 )
                Serial.print(draw, DEC);
            else
                Serial.print(".");
            x++;
        }
        Serial.println("");
        y++;
    }
}

const uint8_t max_player = 10;

void setup() {
```

```
// put your setup code here, to run once:
Serial.begin(115200);
while (!Serial); // Wait for the PC to open the serial port
Serial.println("Serial port has been opened");
delay(5000);
}

void loop() {
  // put your main code here, to run repeatedly:
  static NPC npc[max_player];

  Serial.println("Looping...");
  BOARD().draw(npc, max_player);
  int player = 0;
  while( player < max_player ) {
    npc[player].move();
    player++;
  }
  delay(100);
}
```

Things we are not discussing about classes

Destructors

Function called when a class is destroyed

Protected

Can be accessed by class methods (functions) and friends. Not discussed here.

Function Overloading

Outside the scope of this class. Can be used to pass different number of parameters to different functions with the same name, example:

```
int void add(int a, int b)
```

```
int void add(int a, int b, int c)
```

Inheritance

Outside the scope of this class.

Topic: Libraries

Wiring Libraries

Arduino and chipKIT build on the Wiring Paradigm

Parts of a library

Header File

Source File (may be in the header)

Keywords file

Libraries may be C or C++

Week 0x0D – Liquid Crystal Display (LCD) and Shift Registers

In this lab we are going to add a LCD (Liquid Crystal Display) to our project with utilizing the SPI port. An LCD is a visual output device. LCD's generally come in four flavors of devices: dotmatrix, character (bitmapped dotmatrix per character), segmented and a hybrid displays of the previously mentioned. Displays come in either a mono chromatic or can be colored differently from display to display. There are also many RGB (Red Green Blue) graphical bitmap displays available.

Sample Liquid Crystal Displays



LCD Construction

Each element of an LCD that can be controlled is a pixel. Dotmatrix displays have many pixels that are usually small squares placed close together. Segment display can have many different pixel shapes that can be very either very primitive such as squares or rectangles or can be very organic and flowing in shape.

Each pixel of an LCD typically consists of a layer of molecules aligned between two transparent electrodes, and two polarizing filters. When a voltage is applied to the electrodes it causes the liquid crystals to twist (or untwist). This twisting of the crystal will allow light to either pass or not through the display. There are two primary types of light paths through the display, transmissive and reflective. Reflective displays have a reflective surface under the liquid that light is reflected off of and require lots of ambient light to be able to see the pixels of the display. Transmissive displays allow light to easily pass through them and usually require a backlight in order to see the display. Transmissive displays usually are very hard to read in direct sunlight and reflective displays are very hard to read in a dark room.

On smaller displays such as we will use in this lab, there is one electrode that acts for all pixels in the display and is usually called the backplane electrode. Then each pixel will have its own electrode that allows for individual control. When there are a large number of pixels in a display, it is not practical to drive each pixel directly. This is resolved by multiplexing the pixels in the display and providing different backplane electrodes for blocks of the display.

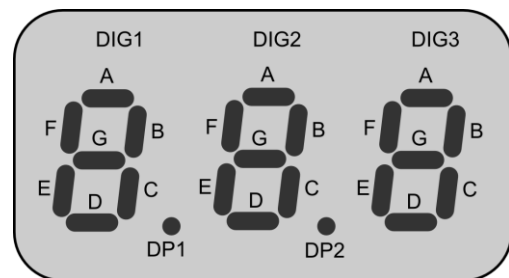
The Big Picture

In this lab we will be interfacing to a three character seven-segment display. Each character in the seven segment display has, you guessed it, seven segments (or pixels). The display also has two segments that are decimal points between the characters.

Wikipedia Information on Electrodes

An electrode is an electrical conductor used to make contact with a nonmetallic part of a circuit (e.g. a semiconductor, an electrolyte or a vacuum). The word was coined by the scientist Michael Faraday from the Greek words *elektron* (meaning amber, from which the word electricity is derived) and *hodos*, a way.

Faraday, Michael (1834). "On Electrical Decomposition". *Philosophical Transactions of the Royal Society*. Archived from the original on 2010-01-17. Retrieved 2010-01-17. (in which Faraday coins the words electrode, anode, cathode, anion, cation, electrolyte, electrolyze)



The display is normally blank, but a pixel will turn opaque when a potential is applied from the backplane to the electrode of the element we wish to display. Only a small voltage is required to cause the pixel to turn opaque and the 3.3V out of a pin of the PIC is plenty to accomplish this. However, in this lab we will not be biasing the LCD with the PIC, but with the output from shift registers.

Biassing the LCD

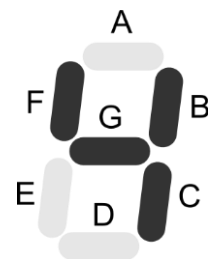
The LCD is not polarized and it does not matter whether we apply 0V to the backplane and 3.3V to the pixel electrode or the other way around (3.3V to the backplane and 0V to the pixel element). In fact, not only does the polarity not matter but it is desirable to alternate the polarity very rapidly. If the polarity is not alternated then the current running through the liquid crystal will cause the crystals to migrate out of their location and the pixels to blur over time. Our driver that writes to the display will handle this by alternating the voltage going to each pixel and the backplane each time we refresh the display.

Bitmapping

In the DAC lab we use an oscilloscope to display images and text. The graphics in that lab were rendered as pixels (a pixel is a picture element) on a dot matrix. We rendered these pixels by plotting a pixel for each x,y location we wanted lit up. Storing information like we did may be desirable if we only have to plot a few points, but if we need to update large portions of the display and may be displaying widely varying images then bitmapping is a solution that we can use to pack a lot of information into a small space. In the case of the two 8-bit x,y dac's we have a the possibility of having 65536 (256x256) pixels. Each pixel can be either on or off (hey that's binary). So if we reserved enough memory to store each of the 65536 bits on or off it would require 8192 bytes (1 bit per pixel, 8 bits per byte so $65536 / 8 = 8192$). This seems like a lot of bytes for our little PIC especially when we are using a 8K byte limited device and indeed it is. The efficiency of storing information in a bitmap only becomes an advantage over x,y locations when at greater than 6.25% of the pixels are lit up ($65536 * 0.0625 = 4096$ pixels, $4096 \text{ pixels} * 2 \text{ bytes per location} = 8192$). Also, once you switch to a bitmap all images have the same storage requirements creating predictable storage requirements. Further, there are many efficient algorithms available to compress a bitmap to an even smaller size (such as RLE (Run Length Encoding)).

Storing Character Maps

When working with the seven-segment displays we will need to turn on the pixels that are needed to make our numbers and alphabet. The pixels in a seven-segment display are labeled a through g in a clockwise rotation around the outside of the character starting at the top segment and ending with the g segment in the middle. Each segment can be either be either on or off (binary). So if we wanted to make the number four we would need to turn on segments b,c,f and g as shown in the illustration.



We could use an x,y type scheme to keep track of which pixels we want on, but since there are so few pixels in this display we would only need one byte and could create a function that would turn on a segment that could look something like this:

```
lcd_segment('B');
lcd_segment('C');
lcd_segment('F');
lcd_segment('G');
```

Although functional, this scheme would require a maximum of 7 bytes to store essentially 7 bits all of which would fit into a single byte. If we bit map the character assigning each pixel to a bit position in a byte then we could store any possible character image in a single byte. I'm going to choose to make the lsb, segment a and assign each remaining segment in increasing order to the next most significant bit leaving bit 7 unassigned to a value. Then if I want a segment lit the bit will have a value of one and not lit will be a value of zero. Thus the bit map for the number four will look like this:

Value	g	f	e	d	c	b	a	Hex
4	0	1	1	0	0	1	1	0

The creation of the remainder of the bitmaps are left as an exercise in the lab.

Waste No Bits

The remaining bit will not go unused. In our display we have three characters and if we have one byte per character this means we will have three unused bits. We shall not let these bits go to waste. Our display also has two decimal points that can be either on or off. The way the display is physically wired these decimal points are associated with the two right most characters and so we will use bit 7 in these two characters as decimal point.

There is also the matter of the backplane. As mentioned previously we will need to alternate the backplane voltage between 0V and 3.3V to prevent liquid crystal migration. Since the left most character still has a free bit 7 we can use this to control the backplane voltage.

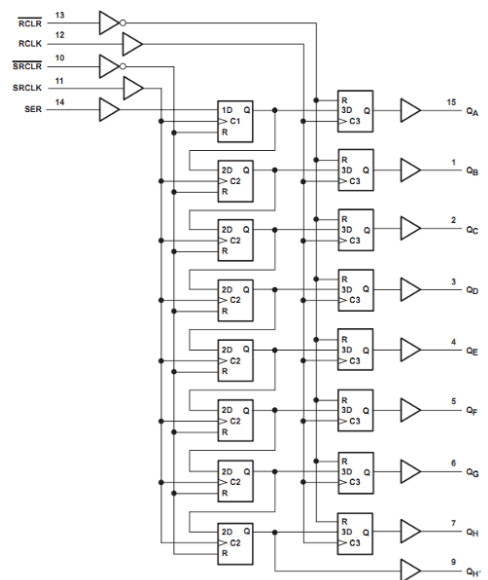
Getting the Data to the Bits

So by now we know we have three bytes worth of data that need to get to the display and if your keeping track of bits this means that there are 24 bits worth of output needed (3 character x 8 bits per character). Since our PIC has only 28 total pins if we dedicated 24 of them to displaying data on the LCD we couldn't do much of anything else (if anything). So rather than tie more processor pins we can just hook up a few shift registers (serial in, parallel out) to the SPI bus and whaw-la, 24 more outputs.

Shift registers can be chained together where the last bit out of the register can be tied to the first bit of the proceeding shift register to create a larger multi-byte shift register.

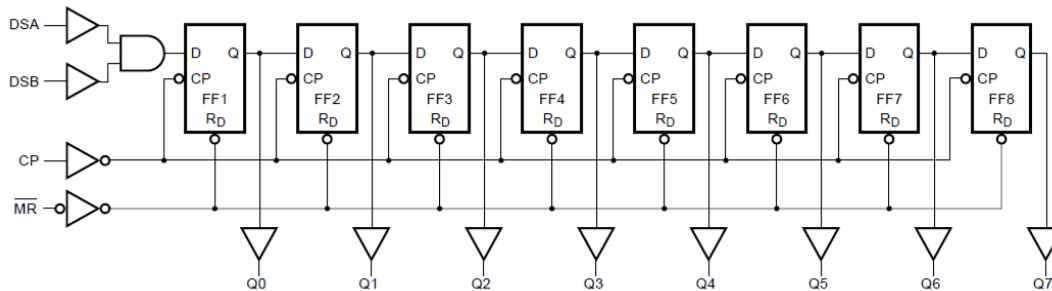
A Funny Thing Happened on the way to Lab

When I was choosing the chips for the labs I was rushed and didn't have as much time as I would have liked. So I ended up choosing a shift register that doesn't quite work the way I would have liked. Some shift registers are double buffered, that is to say data can shift through them without affecting the output. Then when the data is ready to be presented to the output a clock signal moves the data on the internal shift



register to the output registers such as the 74594 shown here.

Not to be discouraged I decided that there was a learning opportunity here, two in fact. The first is that often you need to slow down to get the job the job done correctly from the beginning and that this in the long run will save time. If the PIC we are using didn't have the re-assignable pin feature (and most do not) then the chip chosen would not have worked and we would have needed to use a chip as shown above. The second is the versatility of the re-assignable pin feature is allowing us to get out of a pinch of not having the correct chip.



74164 Functional Diagram

With the shift register that has an output latch the clock and data can always be hooked to the shift register and a chip select need only be used to indicate that the data is ready to be presented to the outputs. With the shift register we have there is no way to disable clocking of data through the chip. So when we are sending data to the DAC this data will be presented on the output of the shift register and therefore on our display. There are several ways we can fix this.

We can buy new chips that work with the SPI model. If we were designing this into a product and may expect to add additional SPI devices later and we had only spent about \$1.50 on the shift register we had then this would probably be the best solution.

We can add external logic that prevents the clock and or data signals from getting to the chip that is controlled with a chip select line. This is nice in that it follows the SPI model and makes the device now work properly, but the external logic chips will add size and cost to the circuit. This option may be best if there is little time or money and you have the chips on hand to solve the problem. When I say little time I mean less than 24-48 hours since you can get replacement chips overnight from a distributor such as Digikey and when I say little money I mean less than a few dollars plus shipping costs.

Or we can use the re-assignable pin feature of the PIC and take the clock signal away from the device when we are not shifting bits out the port.

We will choose the latter.

Shifting Through The Registers

When hooking up three shift registers,

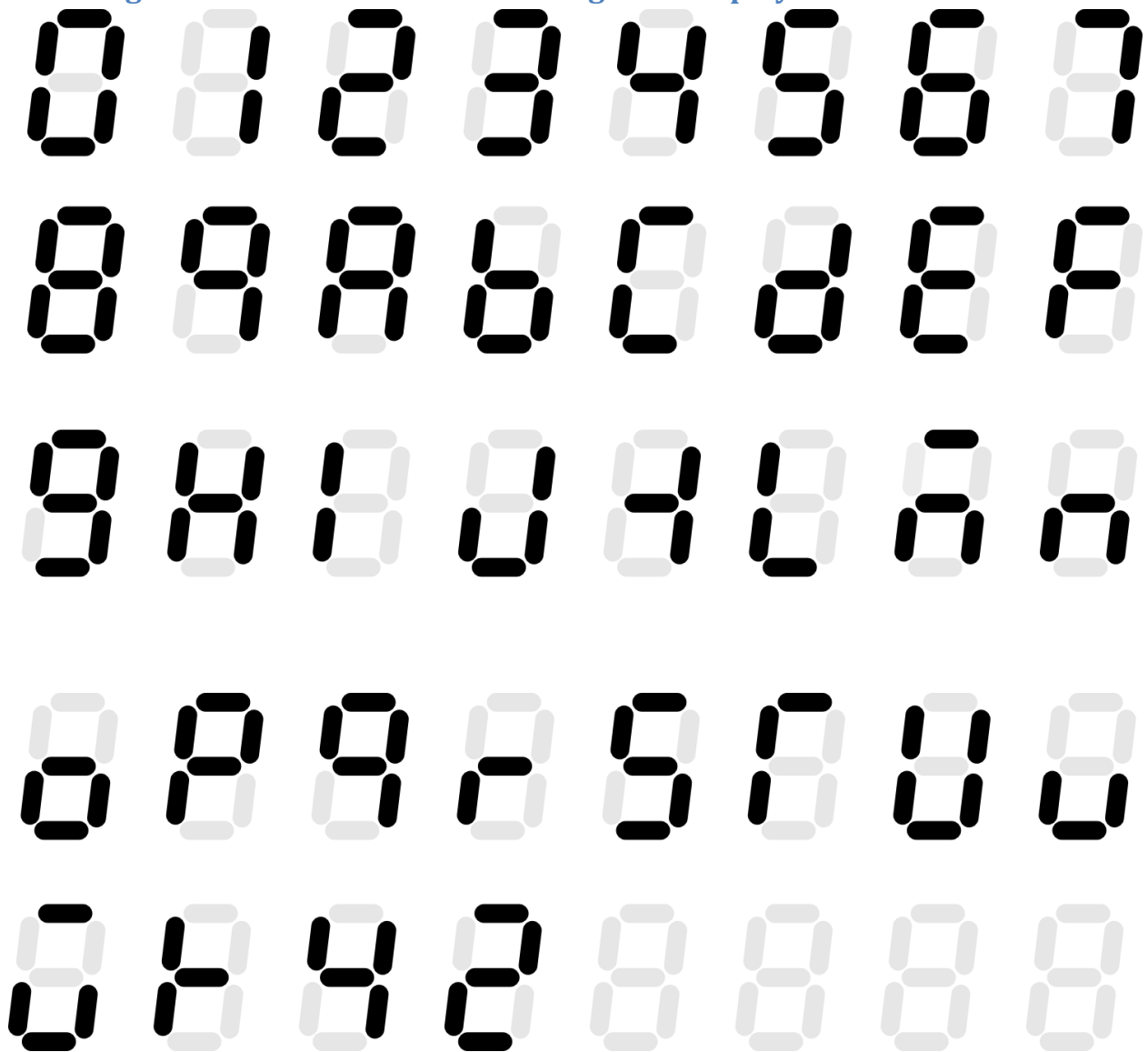
API

The final API for using this LCD might look something like this:

```
lcd_display_number(uint16_t value, uint8_t dp_bits);  
lcd_display_string(char* string, uint8_t dp_bits);
```

Where value the first function will display a numerical value between 0 and 999. And the second will display a three character string.

Encoding Numbers and Letters on a 7-segment display



Week 0x0E - Tones, Timing, Time, PWM, Servos and Tasks

Tones

The `tone()` and `noTone()` are used to control generation of a square wave output on a single pin. They are named `tone` and `noTone` because they are commonly used as outputs to drive speakers or piezo buzzers to play tones. They could also be used for other applications that require a variable frequency input.

`tone()`

Generate a tone (square wave) on an I/O pin at frequency Hz. If duration is greater than 0 the tone will play for the specified number of milliseconds. If duration is less than 0 it will be generated indefinitely until `noTone()` is called. If duration is 0 it acts the same as if `noTone()` had been called.

Prototype(s)

```
void tone(uint8_t pin, unsigned int frequency)
void tone(uint8_t pin, unsigned int frequency, unsigned long duration)
```

Parameter(s)

pin: GPIO pin to generate square wave on.
frequency: Frequency in Hz of generated square wave.
duration: time in milliseconds to play the tone, 0 will play indefinitely.

Return Value

None.

Example(s)

Several examples are available in the File->Examples->02.Digital section of the Arduino IDE. All examples that start with the word `tone` use these functions.

`noTone()`

`noTone()` stops generating a tone on an I/O pin which has previously been started through the `tone()` function.

Prototype(s)

```
void noTone(uint8_t pin)
```

Parameter(s)

pin: GPIO pin to stop generating square wave on.

Return Value

None.

Example(s)

Several examples are available in the File->Examples->02.Digital section of the Arduino IDE. Some examples that start with the word `tone` use this function.

Timing

To time a fast changing pulse on a specific pin the `pulseIn()` function can be used.

Time

The following functions exist that are helpful for creating timed delays or timing events:

`delay()`, `delayMicroseconds()`, `millis()` `micros()`

PWM

There is also another set of functions that can be used to generate square waves with variable duty cycle called `analogWrite()`. The `analogWrite()` functions also have are so named because they if the digital pulse signal is integrated (averaged) then the duty cycle becomes the average voltage generated (similar to using a DAC).

`analogWrite()`

`analogWrite()` is used to control the PWM functionality of pins which are connected to (or can be routed to) the hardware Output Compare module within the main MCU. Pins which do not support PWM will be either set to a constant HIGH or LOW value depending on the value input to the function. A 50% threshold value is used to decide if the pin should be HIGH or LOW.

The PWM generated is at an 8-bit resolution and typically in the region of 500Hz carrier frequency. The input value 0 and 255 are equivalent to a constant LOW and constant HIGH signal respectively. Values from 1 to 254 inclusive represent a duty cycle for the generated square wave. The percentage value of the duty cycle is equivalent to $(val \div 255) \times 100$ so 127 represents a 50% duty cycle.

Prototype

```
void analogWrite(uint8_t pin, int val)
```

Return Value

None

`analogWriteResolution()`

`analogWriteFrequency()`

Servos

The servo library allows you to generate a PWM signal specifically for controlling an RC servo.

```
#include <Servo.h>

Servo myservo;

void setup()
{
  myservo.attach(9);
  myservo.write(90); // set servo to mid-point
}
```

```
}  
  
void loop() {}
```

Tasks

The following documentation is specifically for chipKIT-core.

The task manager allows the creation and operation of background tasks. A task is a user defined function that is executed automatically at specified intervals or at a specified time. Using tasks can simplify the logic of programs that need to perform periodic operations.

Scheduling of tasks for execution is done prior to each execution of the user function `loop()` and during time spent in the `delay()` function. In order for task scheduling to work reliably, the time to execute the user `loop()` function should be short; preferably less than a millisecond, but no more than a few milliseconds if the timing accuracy of task scheduling needs to be precise. Time spent in task functions contributes to the total execution time of the `loop()` function, as all task functions currently scheduled for execution will be called before `loop()` is called.

The actual time at which a task will be run can vary by $-0/+N$ milliseconds from the scheduled time, where N is the longest time that it takes for execution of the user `loop()` function plus the longest execution time for all tasks that can be scheduled for execution at the same time. The interval from when a task is created or enabled to the time of first execution can be ± 1 millisecond in addition to the above noted error due to the fact that the user program's execution is asynchronous to when the millisecond tick counter is updated.

The scheduling times for task functions is based on the system millisecond tick counter. This is the value returned by the `millis()` function. This tick counter runs continuously and gives the number of milliseconds elapsed since the system started running.

Here is a sample blink sketch using the task manager:

```
/*  
  Blink Task  
  Blink LED1 with a 500ms cycle (2Hz)  
  Blink LED2 with a 510ms cycle (1.96Hz)  
  Utilizing the chipKIT task manager rather than delay functions.  
  This example code is in the public domain.  
*/  
  
int blink1_id;  
int blink2_id;  
unsigned long blink1_var;  

```



```
void blink_task1(int id, void * tptr) {
    digitalWrite(PIN_LED1, !digitalRead(PIN_LED1)); // Toggle pin state
}

void blink_task2(int id, void * tptr) {
    digitalWrite(PIN_LED2, !digitalRead(PIN_LED2)); // Toggle pin state
}

void setup() {
    // initialize the digital pin as an output.
    // Pin 13 has an LED connected on most Arduino boards:
    pinMode(PIN_LED1, OUTPUT);
    pinMode(PIN_LED2, OUTPUT);
    blink1_id = createTask(blink_task1, 250, TASK_ENABLE, &blink1_var);
    blink2_id = createTask(blink_task2, 255, TASK_ENABLE, &blink2_var);
}

void loop() {
}
```

Week 0x0B / Fubarino SD, SD Cards

Accessing prior information

What is an SD Card

Byte addressable non-volatile memory in a small removable package.

Flash Memory

Non-volatile

Writable and erasable memory

Stores Information

files and folders

meta data

File name

Size

Last touched (modified) date time

Location on disk

Permissions

Directory Structure

File system Information (FAT)

Volume

Digital interface for accessing information

SPI

1-Bit SD Bus Mode

4-Bit SD Bus Mode

File System

The file system is an arbitrary construct that is imposed on the memory of the storage device to organize the information stored within. It is arbitrary in that there are many to choose from (in the world). Though there are tradeoffs between the selections of one verse another almost all have a primary user perspective in play. Namely a structure to store information about the information stored on the device.

The default file system on most SD cards when purchased is some form of FAT (File Allocation Table). If the file system on the card is not what you need, it can be changed by a program capable of such functions.

Hierarchy

This is a short, but popular, list of existing file systems in use:

FAT16, FAT32, NTFS, ext2, ext3, ext4, HFS, HFS+, ZFS

Why use a file system?

You can transfer files to and from another device that recognizes the file system.

What is a file

Name

The file name is the text presented to the user of a file system that is used to uniquely identify a specific file in a specific directory. It is therefore not possible to have two or more files with exactly the same name in the same directory. Modern operating systems support so called "long filenames" because the names of the files can be quite long (255 characters). This, however, was not always the case. Prior to Windows 95, MS-DOS and Windows computers only supported "8.3" filenames on a FAT file system. Since most SD cards are formatted with the FAT file system and since there are patents associated with long file names and FAT, chipKIT and Arduino currently only support 8.3 filenames. The 8.3 filename can have a maximum of eight characters followed by a period then a maximum of three more characters.

Data

All files are best thought of as an ordered sequence of binary data. The data in its simplest form is an ordered sequence of bytes and would be considered a binary file. Files are categorized into different file types, but all file types at their core are just an ordered sequence of bytes.

File types

Early on in the computing world, files fell primarily into two types. Binary and text. As previously stated, all files are really binary files, so what differentiated a text file from a binary file was that data stored in the text file is interpreted as ASCII data. Interpreted by whom? By whatever program was reading the file. The primary way a program knows what the file type is by means of an extension that is added to the end of the file.

Timers / Real Time Clock Calendars

The connotative meaning for the general public of a timer will invoke ideas of a egg timer like that which is used in a kitchen or a stop watch such as is used in sports. However when we use the term timer in the embedded systems context we are typically thinking of something very different yet related to the afore mentioned devices. The egg timer and stop watch are examples of real time clocks.

A real time clock or RTC as they are often referred to maybe better thought of as clocks that work on a human time scale (that is time that is "real" to a human). Where as a timer in an embedded systems timers is used to time events in sub human time scales. For a PIC32 (chipKIT) running at 80MHz

Fubarino SD uC's

PIC32MX440F256H and PIC32MX795F512H

Core Timer Service Overview

(from chipKIT.org)

The core timer is a facility built into the MIPS M4K processor core in the PIC32 microcontroller. It is made up of a 32-bit counter register and a 32-bit compare register. The counter register increments at 1/2 the processor clock frequency (SYSCLK). The default SYSCLK frequency is 80Mhz, so the core timer counter increments at 40Mhz. The compare register can be used to trigger an interrupt (core timer interrupt) when the counter matches the value loaded into the compare register.

In the chipKIT Arduino IDE system, the core timer is used to manage the timing of events at 25ns (nanosecond) resolution. The core timer service facility allow service functions to be registered that will be called by the core timer interrupt service routine (ISR). A core timer service function indicates to the core timer ISR the system time (i.e. core timer counter value) at which it should be called next. This allows a service function to schedule itself to be executed at any time up to 90 seconds in the future with 25ns resolution. This allows event timing to be done with great precision.

In the chipKIT Arduino IDE system, the core timer service facility is used to implement the low level timing function millis(). When the system starts up, it registers a core timer service function that schedules itself to be called once each millisecond. This service function then maintains the system millisecond tick counter that is returned by the millis() function.

A core timer service function is a callback routine that is registered with the CoreTimerHandler Interrupt Service Routine (ISR). This function will be called when the core timer counter has reached the core timer service's trigger time. Each time a core timer service function is called, the current core timer counter value is passed in as a parameter. The core timer service function returns the counter value of the next time it wishes to be called, i.e. the next trigger time.

The core timer service function is guaranteed to be called no earlier than it's next scheduled time, but may be called late. This can occur if interrupts have been disabled, as the core timer ISR will not execute again until interrupts are re-enabled. This will also occur when writing to the flash memory in the PIC32

microcontroller. When a flash memory page write is performed, the processor stops executing instruction for 20ms, although the core timer counter continues to increment. In general, the current time passed in will typically be a few ticks after the requested trigger time. This is usually not a problem as the tick period is 25 nsec. However, if interrupts have been disabled, the service function may be called as much 20-50ms late. It is up to the service function to handle being called late.

The core system time (i.e. core system counter value) is a 32 bit unsigned integer and wraps once every $2^{32} / 40,000,000$ or ~ 107.3741824 seconds. When the service function is called, the current time as represented by this 32 bit unsigned integer is passed in as a parameter. Up to 90 seconds may be added to the current time to specify the next trigger time. Do not worry about the 32 bit unsigned integer wrapping in value as the core timer service assumes the next 90 seconds of time is in the future. Do not exceed 90 seconds as the CoreTimerHandler potentially regards anything beyond that is a time before the current time. It is a requirement that a core timer service function return a next trigger time 0-90 seconds in the future. It may not return something that the CoreTimerHandler ISR may regard to be in the past, that is, do not subtract from the current time, always add to it.

When a core timer service function is registered, the first call to the function will occur on the next regularly scheduled call to the CoreTimerHandler ISR. The system always has a millisecond CoreTimer Service registered and this will typically ensure that a newly registered Service will be called within 1 ms of being registered. However, should interrupts been disabled, this may be late up to 20-50ms.

Currently, a maximum of 3 core timer services functions can be registered simultaneously. One is always taken by the millisecondCoreTimerService. Therefore there are two available slots open for use. To register a core timer service function, call `attachCoreTimerService()` passing a pointer to the service function. The `attachCoreTimerService()` function will return false (0), if there are no open slots available. You may de-register (remove) a core timer service function by calling `detachCoreTimerService()` passing a pointer to the function to remove. Once removed the slot becomes available for another core timer service function to be registered. NEVER remove the system millisecondCoreTimerService function.

The rules for a core timer service function:

Do NOT set the core timer "compare" register directly! If you don't know what this is, GOOD, don't fool with it.

Do not do anything that could cause the CoreTimerHandler ISR to be called recursively. Primarily, this means do not enable interrupts as the core timer interrupt flag is still set and will immediately cause the system to call CoreTimerHandler ISR recursively.

The current time is passed to the service function as a parameter. This is usually several ticks after the requested trigger time, but if interrupts were disabled this may be as much as 20-50ms late.

The current system time is obtained by reading the core timer counter register when the CoreTimerHandler ISR is entered. This is the value passed as a parameter to the service function. It may actually be several ticks old. For this reason is okay to read the core timer counter register directly.

Typically this is not necessary as the current time is only out-of-date by a few nsec, which is just the delay in the instructions executed to call the callback Service.

A Service will never be called before the requested trigger time, but it may be called late.

Do not return a next trigger time more than 90 seconds in the future. Each tick is 25 nsec, there are 40,000,000 ticks in a second; therefore do not add more than $90 \times 40,000,000$ to the current time for the next trigger time. Do not attempt to return a negative time, always add time to the current time, do not subtract. It is okay for the 32 bit unsigned integer value returned to wrap when added to the current system time to determine the next trigger time.

If the service function returns a next trigger time that is very near to the current time, it is possible for that system time to have already passed. Under this condition, the CoreTimerHandler will immediately call the service function again with an updated current time without exiting the CoreTimerHandler ISR.

Because the CoreTimerHandler ISR may immediately call a service function without exiting the ISR (as in the previous rule), it is imperative that the service function execute in a timely manner. If the service function takes too long to execute, it is possible to create a condition where the CoreTimerHandler ISR is never exited. In this case, no processor time will be given to execute anything else, include the primary sketch. Remember, the service function is executing within the context of an interrupt, so the function's code should be written to complete very quickly.

Once a Service is registered, it is typically called for the first time within 1 ms of registration unless interrupts were disabled, and then it could be as much as 20-50ms late.

Do NOT remove the pre-registered millisecondCoreTimerService CoreTimer Service, this will break the system!

Examples:

Example 1

The simplest of core time service examples: simply schedule a callback at a particular frequency (in this case, 10KHz) and toggle an output pin in the callback. This callback produces a 500nS pulse on pin 4 every 100uS.

```
/* CoreTimer demo1 : demonstrates a simple callback scheduled at
a single frequency (10Khz). This example code is in the public domain. */

void setup() {
  pinMode(4, OUTPUT); // Use IO pin 4 to show operation of callback
  attachCoreTimerService(MyCallback);
}
```

```
// We don't need to do anything in the main loop
void loop() {
}

// For the core timer callback, just toggle the output high and low
// and schedule us for another 100uS in the future. CORE_TICK_RATE
// is the number of core timer counts in 1 millisecond. So if we
// want this callback to be called every 100uS, we just divide
// the CORE_TICK_RATE by 10, and add it to the current time.
// currentTime is the core timer clock time at the moment we get
// called.
uint32_t MyCallback(uint32_t currentTime) {
    digitalWrite(4, HIGH);
    digitalWrite(4, LOW);
    return (currentTime + CORE_TICK_RATE/10);
}
```

Week 0x10 - Mid Term 2 (Final)

Errata's

Don't forget to read the chip errata's (especially before you commit to a chip for your design).

Appendix A – History chipKIT and the development tools

Arduino got its start by combining the MIT Wiring library (<http://wiring.org.co/>) with a GUI that is based on the Processing programming language (<https://processing.org/>).

The Arduino project started in 2005 and was an early open hardware project that gained momentum quickly and is seemingly everywhere as of this writing in 2016.

On May 23rd of 2011 Microchip Technology and Digilent Inc. announced the release of chipKIT UNO32 and MAX32 development boards based on the Microchip PIC32 Microcontrollers. The original announcement stated that chipKIT was software and hardware compatible. Unfortunately, the compatibility was not 100% and there was community backlash.

The original Arduino IDE that supported only Atmel processors is forked by Rick Anderson and Mark Sproul to create MPIDE with the goal of creating a development tool that is Arduino compatible but would work for any embedded platform. They fell short of their goals but did get it to work for Microchip PIC32 processors.

October 22nd 2012 Arduino Due is released. Changes made in MPIDE to support PIC32 were taken back into the Arduino IDE tool to allow the ARM base Arduino Due to compile in the Arduino IDE.

December 12th 2016 Arduino IDE version 1.6.7 has ability to take a link to import cores for other processors.

January 19th 2016 chipKIT-core released with ability be used within Arduino 1.6.7

Sources

<http://chipkit.net/about-us/>

<https://www.arduino.cc/en/Main/ReleaseNotes>

<https://en.wikipedia.org/wiki/Arduino>

<http://www.microchip.com/pagehandler/en-us/chipKIT-Development-Platform.html>