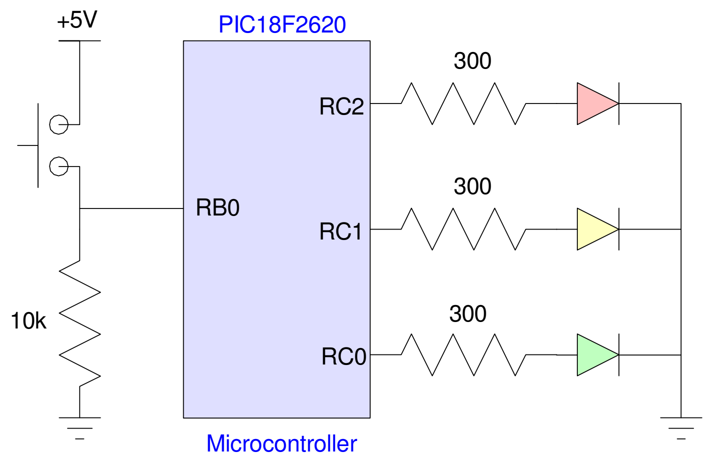
Designs using a PIC18F4620 and C



In Senior Design I, many of the projects can be built just using digital logic and 555 timers. They could also be built *using* a microcontroller.

Microcontrollers are just a tool: if the tool helps you do your job, use it. If not, don't use it. If you don't use a microcontroller, you don't need to worry about

- Designing hardware around the microcontroller,
- Having to write and debug code, and
- How to download that code.

If you are willing to learn how to do this, however, microcontrollers can give you a great deal of flexibility in your design.

In this lecture, we going to cover

- Hardware: How to wire up a PIC chip so that you can make a light blink
- Downloading: How to get your code onto the PIC chip, and
- Coding: How to write simple C routines to make a light blink

I like to say that only engineers get excited when a light blinks. Getting a light to blink is a big deal. A blinking light means

- You were able to compile your code
- You were able to download your code, and
- Your code is running.

Once you get a light to blink, the rest is easy (sort of)...

Hardware

There are tons of microcontrollers out there. In Senior Design I, only the PIC18F2620 is allowed for several reasons:

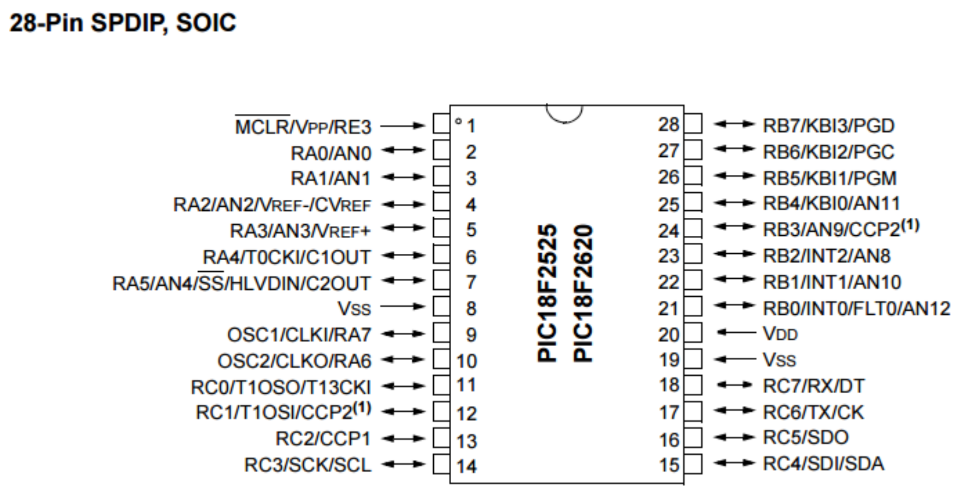
- You can find pre-written code online for just about everything for an Arduino and Raspberry Pi. A degree in ECE should mean more than you know how to search the web.
- We have a boot-loader for this chip (same as used in ECE 376)
- We have experience using this chip (same as ECE 376)
- We have a C compiler for this chip (same as ECE 376)

The only difference between the 40-pin version used in ECE 376 and the 28-pin version used in ECE 401 is

- You have 22 I/O pins with the 28-pin version (vs. 33 I/O pins), and
- PORTD and PORTE are not connected to any I/O pins with the 28-pin version

Otherwise, they're the same.

If you look up the data sheets for a PIC18F2620, the I/O pins can be found.

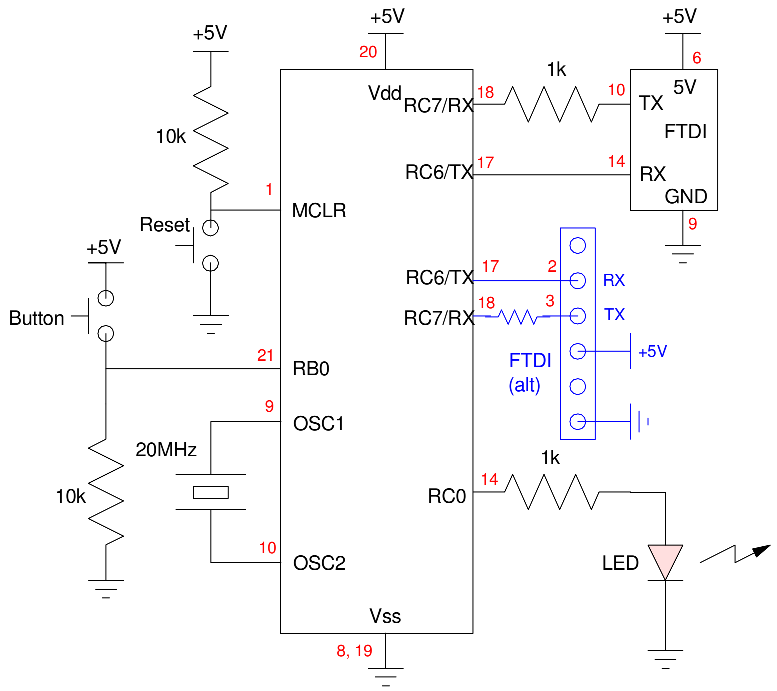


Pinouts for a PIC18F2620

When you design a system around a PIC processor, you need to identify the function of each I/O pin. With this processor, there are three I/O ports: A, B, and C. If a single LED is connected to PortC pin 3, the pin assignments could be something like this:

PORTC							
7	6	5	4	3	2	1	0
TX	RX	-	-	LED	-	-	-
Out	In			Output			

A schematic for this minimal setup:



Schematics for getting a PIC to run and drive an LED on PORTC pin 3.
The FTDI can be connected using the 18 pins around the edge as shown in the photo below
It can also be connected using the six connections at the edge of the board (shown in blue above)

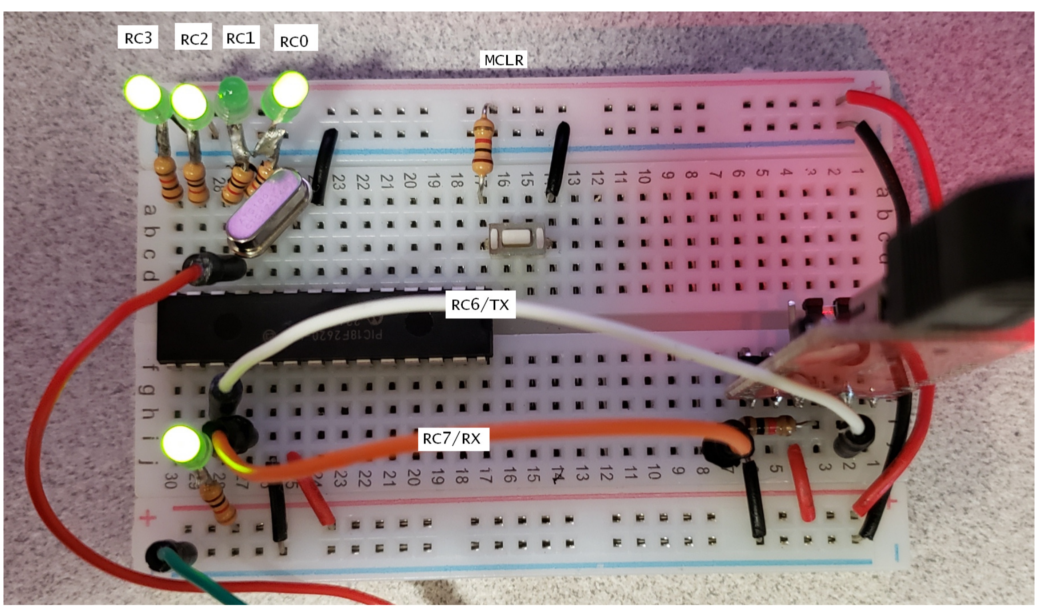


Photo of breadboard for the above schematic

Compiling C Code & Using MPLAB8

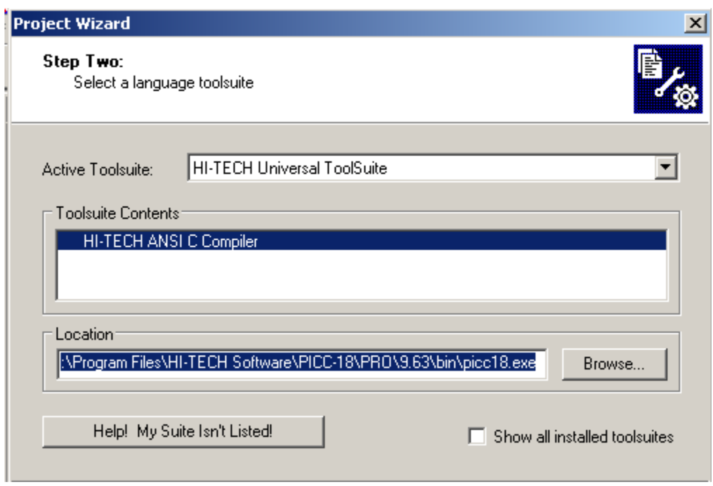
Step 1: Start with a working program. Typically, open a zip file and copy all of its contents to your z-drive. I'd recomment something like

z:\ECE401\Clock

Step 2: Start MPLAB. Go to the program wizard (just like you did in assembler)

Select your device: PIC18F2620 (or 4620)

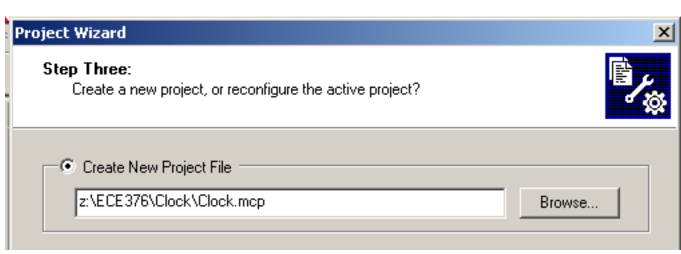
Select the Hi-Tech C Universal Toolsuite.



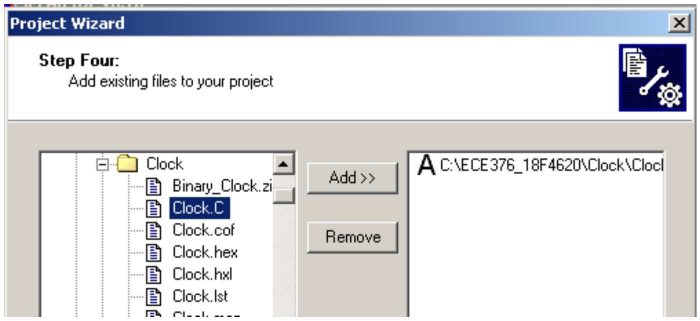
This tells the compiler to interprit your code as C code. Note that if this isn't an option under the Active Toolsuite, there's a problem. This usually means the C compiler is in a read-only directory and needs the permissions changed by a system administrator.

Assuming that works...

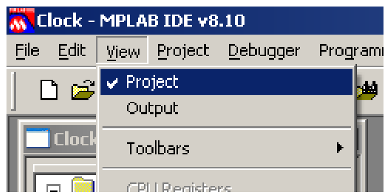
Change the path to your z-drive for where the files are located



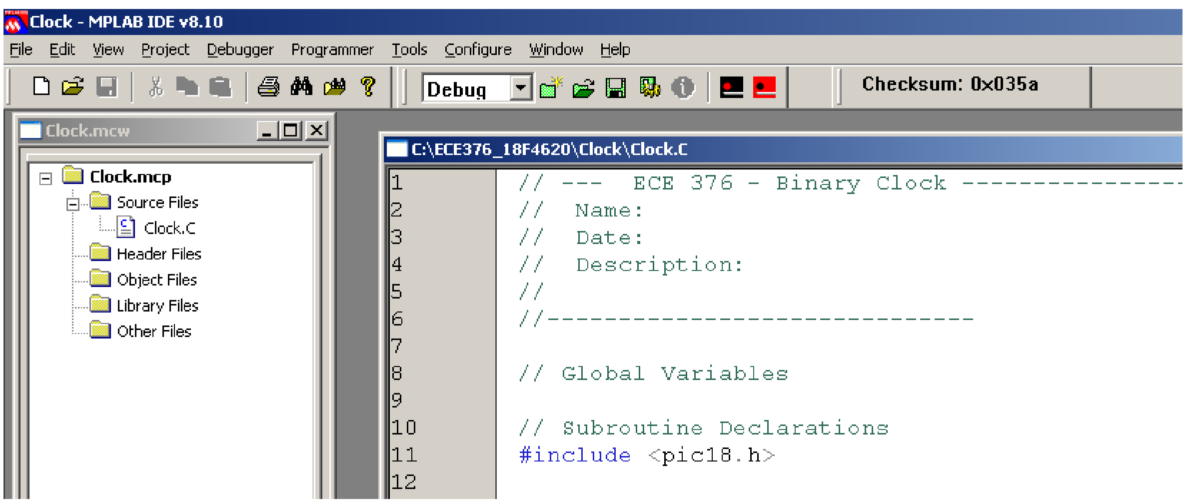
Select the C program you want to compile (usually the name of the directory)



You should get the following screen. If not, select View Project



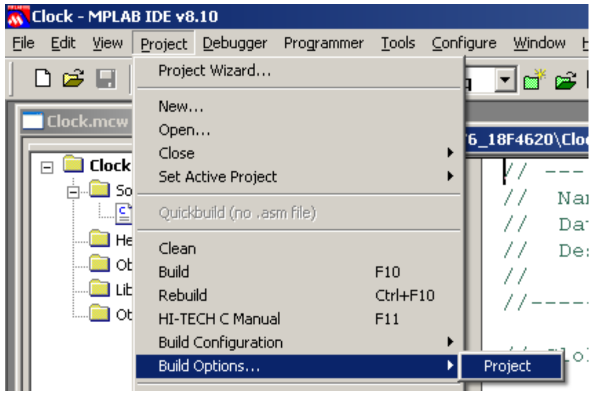
You should get the following screen:



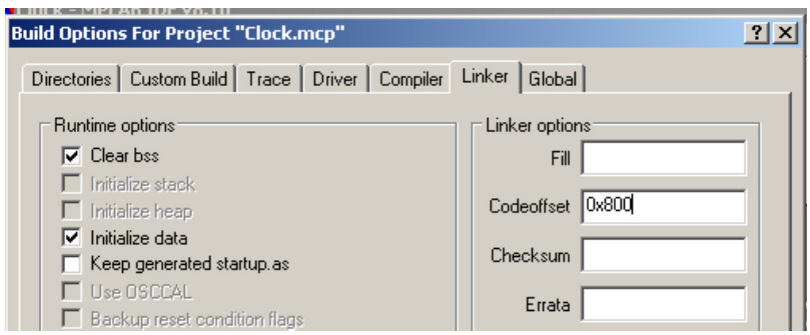
*** important *** Offset your code by 0x800

Your code needs to start at 0x800 - after the boot-loader.

Go to Project - Build Options - Project



Under Linker, offset the code by 0x800



note: If your code worked yesterday and doesn't work today, it's probably you forgot to offset your code by 0x800

Compile y our code just like you did in assembler

Project Build All (or F10)

You should get the following message

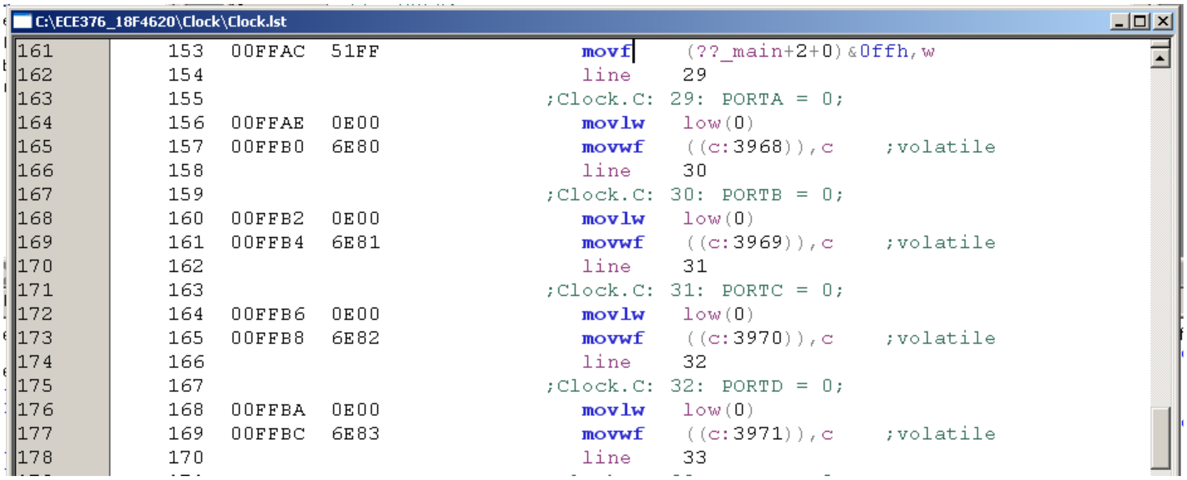
```
Memory Summary:
Program space        used    76h (   118) of 10000h bytes   (  0.2%)
Data space           used     3h (     3) of   F80h bytes   (  0.1%)
EEPROM space         used     0h (     0) of   400h bytes   (  0.0%)
ID Location space    used     0h (     0) of     8h nibbles (  0.0%)
Configuration bits   used     0h (     0) of     7h words   (  0.0%)
```

This tells you your code compiled and uses up 118 bytes (out of 64k), 3 bytes of RAM (out of 4k), etc.

This also creates some files

Clock.lst

This shows how your C code converts to assembler. A section looks like the following



C:\ECE376_18F4620\Clock\Clock.lst

```
153   00FFAC   51FF           movf    (??_main+2+0)&0ffh,w
154                           line    29
155                        ;Clock.C: 29: PORTA = 0;
156   00FFAE   0E00           movlw   low(0)
157   00FFB0   6E80           movwf   ((c:3968)),c    ;volatile
158                           line    30
159                        ;Clock.C: 30: PORTB = 0;
160   00FFB2   0E00           movlw   low(0)
161   00FFB4   6E81           movwf   ((c:3969)),c    ;volatile
162                           line    31
163                        ;Clock.C: 31: PORTC = 0;
164   00FFB6   0E00           movlw   low(0)
165   00FFB8   6E82           movwf   ((c:3970)),c    ;volatile
166                           line    32
167                        ;Clock.C: 32: PORTD = 0;
168   00FFBA   0E00           movlw   low(0)
169   00FFBC   6E83           movwf   ((c:3971)),c    ;volatile
170                           line    33
```

Clock.hex

This is the machine code you download to your processor

```
:04000000C7EF7FF0D7
:10FF8E00000E926E000E936E000E946E000E956E25
:10FF9E00000E966E0001FF6F0F0EC16E0001FF5135
:10FFAE00000E806E000E816E000E826E000E836E4D
:10FFBE00000E846E000E00010001FD6F000E0001A8
:10FFCE00FE6F010E00010001FD2500010001FD6F15
:10FFDE00000E00010001FE210001FE6FFDC083FF37
:10FFEE00836601D001D002D08228826EEAD700EF5C
:02FFFE0000F011
:00000001FF
```

Note that the reason we like C so much is

- It compiles to assembler fairly directly
- Meaning it is efficient, and
- C has things like multiply, divide, loops, arrays.

C-Coding

Once you have the hardware and MPLAB8 compiler ready, you can start coding. Each pin can be input or output

- Input: Read the buttons or other devices.
  - *5V = logic 1*
  - *0V = logic 0*
- Output: Drive something like an LED
  - *Logic 1 = 5V*
  - *Logic 0 = 0V*

The program has to tell the PIC which it is. These are the *TRIS* registers where each bit determines the I/O status of each pin. For example

```
TRISA = 0x00;
```

tells the PIC that all pins on PORTA are output (a zero is written to each bit of TRISA)

```
TRISB = 0xFF;
```

tells the PIC that all pins of PORTB are input (a one is written to each bit of TRISB)

```
TRISC = 0x0F;
```

tells the PIC that the first four pins of PORTC are output (0) and the last four pins are input (1).

The I/O ports can be addressed using their name

```
PORTA = 0x00;     all pins on PORTA are 0V
PORTB = 0xFF;     all pins on PORTB are 5V
PORTC = 0x01;     pin #0 is 5V, the rest are 0V
```

You can also address each bit of a given port

```
RA0 = 1;          Port A bit #0 is 5V, other pins are unchanged
RB3 = 0;          Port B bit #3 is 0V
RC7 = 1;          Port C bit #7 is 5V
```

Also also, you need to include the code

```
ADCON1 = 0x0F;
```

to use binary inputs and outputs. For more details on this, please refer to ECE 376 on analog inputs and outputs.

Program #1: Write 1, 2, 3 to Port A, B, C

C-Code

```
// Subroutine Declarations
#include <pic18.h>

// Subroutines

// Main Routine

void main(void)
{

   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   ADCON1 = 0x0F;

   PORTA = 1;
   PORTB = 2;
   PORTC = 3;

   while(1);

}
```

Compilation Results:

```
Memory Summary:
    Program space        used    2Eh (    46) of 10000h bytes   (  0.1%)
    Data space           used     1h (     1) of   F80h bytes   (  0.0%)
    EEPROM space         used     0h (     0) of   400h bytes   (  0.0%)
    ID Location space    used     0h (     0) of     8h nibbles (  0.0%)
    Configuration bits   used     0h (     0) of     7h words   (  0.0%)
```

This C code compiles into 23 lines of assembler (46 bytes: each instruction is two bytes)

Note: The while(1); statement at the end is a *stop* command. If you remove it, the program will execute until it gets to the end of memory (32k instructions later) then it restarts at address 0x0000, which is where the boot-loader is located.

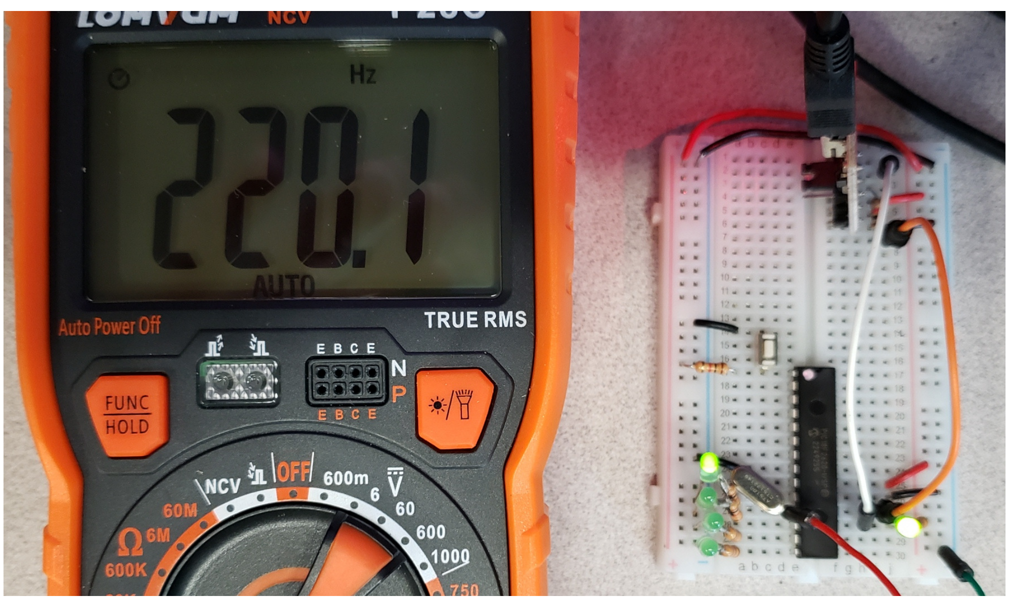
Program #2: Make RC0 blink at 220Hz

```
#include <pic18.h>

void main(void)
{
   unsigned int i;

   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   ADCON1 = 0x0F;
   PORTC = 0;

   while(1) {
      RC0 = !RC0;
      for(i=0; i<1419; i++);
      }
}
```

The compilation results are:

```
Memory Summary:
    Program space        used    6Ch (   108) of 10000h bytes   (  0.2%)
    Data space           used     3h (     3) of   F80h bytes   (  0.1%)
    EEPROM space         used     0h (     0) of   400h bytes   (  0.0%)
    ID Location space    used     0h (     0) of     8h nibbles (  0.0%)
    Configuration bits   used     0h (     0) of     7h words    (  0.0%)
```

The number *1419* is found using trial and error. It sets up a wait routine to set the frequency to 220Hz

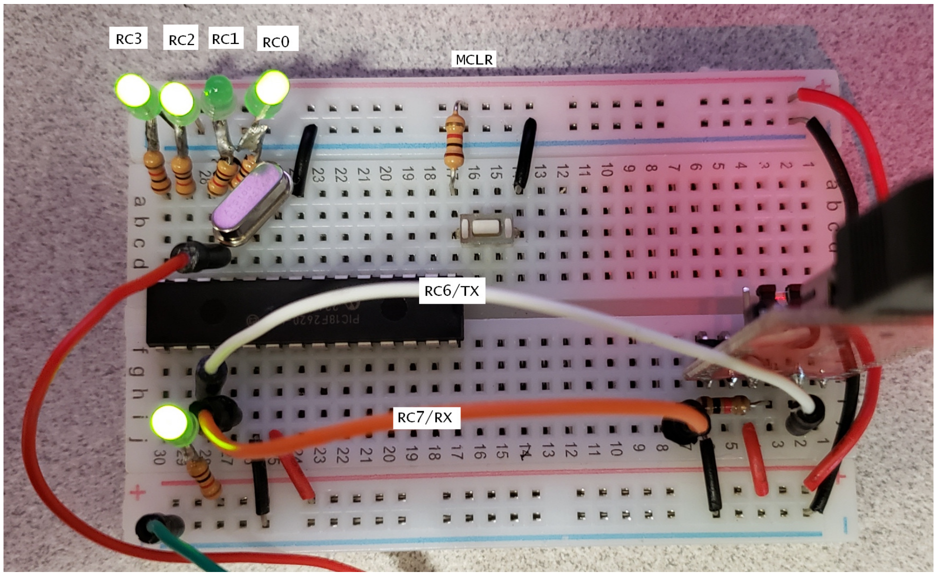


Actual frequency output on RC0 is 220.1Hz

Program #3: Subroutines and Wait loops

Another nice feature of C is you have access to subroutines. Suppose you want to write a routine which counts once per second. One way to do this is create a suboutine, *Wait()*, which waits N milliseconds. The number 617 is found using trial and error: whatever it takes so that Wait(1000) waits 1000ms.

```
// Subroutine Declarations
#include <pic18.h>

// Subroutines
void Wait(unsigned int X)
{
   unsigned int i, j;
   for (i=0; i<X; i++)
      for (j=0; j<617; j++);
   }


// Main Routine

void main(void)
{
   TRISA = 0;
   TRISB = 0;
   TRISC = 0;
   ADCON1 = 0x0F;
   PORTC = 0;

   while(1) {
      PORTC += 1;
      Wait(1000);
      }
   }
```



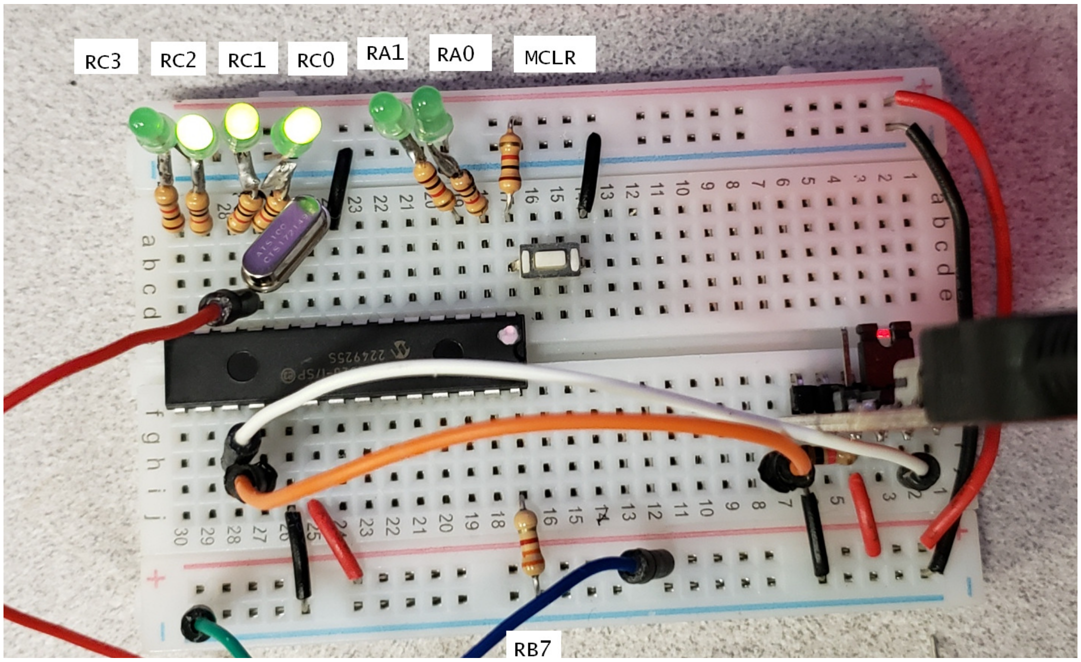
Counting once per second. Current count is 13 (1101)

Program #4: Counter

Beep every time button RB0 is pressed and released

After 10 button presses, turn on the light on RC0 for one second

```
// Subroutine Declarations
#include <pic18.h>

// Subroutines
void Wait(unsigned int X)
{
   unsigned int i, j;
   for (i=0; i<X; i++)
      for (j=0; j<617; j++);
   }

void Beep(void)
{
   unsigned int i, j;
   for (i=0; i<50; i++) {
      RA1 = !RA1;
      for (j=0; j<200; j++);
      }
   }


// Main Routine

void main(void)
{
   unsigned int COUNT;

   TRISA = 0;
   TRISB = 0xFF;
   TRISC = 0;
   ADCON1 = 0x0F;

   COUNT = 0;

   while(1) {
      while(RB7);
      while(!RB7);

      Beep();

      COUNT += 1;
      PORTC = COUNT;

      if (COUNT >= 10) {
         RA0 = 1;
         Wait(1000);
         RA0 = 0;
         COUNT = 0;
         PORTC = COUNT;
         }
      }
   }
```



Counting rising edges on RB7

RB7 is tied to ground through a 3.3k resistor (somewhat arbitrary)

When RB7 is connected to +5V, PORTC counts and a beep is sent to RA1

After 10 counts, RA0 goes high for one second

C Language Summary

Character Definitions:

Name	bits	range
char	8	-128 to +127
unsigned char	8	0 to 255
int	16	-32,768 to +32,767
unsigned int	16	0 to 65,535
long	32	-2,147,583,648 to +2,147,483,647
unsigned long	32	0 to 4,294,967,295
float	32	3.4e-38 to 3.4e38
double	64	1.7e-308 to 1.7e+308
long double	80	3.4e-4932 to 3.4e+4932

Arithmetic Operations

Name	Example	Operation
+	1 + 2 = 3	addition
-	3 - 2 = 1	subtraction
*	2 * 3 = 6	multiplication
/	6 / 3 = 2	division
%	5 % 2 = 1	modulus
++	A++	use then increment
	++A	increment then use
--	A--	use then decrement
	--A	decrement then use
&	14 & 7 = 6	logical AND
|	14 | 7 = 15	logical OR
^	14 ^ 7 = 9	logical XOR
>>	14 >> 2 = 3	shift right. Shift in zeros from left.
<<	14 << 2 = 56	shift left. Shift zeros in from right.

Defining Variables:

int A;	A is an integer
int A = 3;	A in an integer initialized to 3.
int A, B, C;	A, B, and C are integers
int A=B=C=1;	A, B, and C are integers, each initialized to 1.
int A[5] = {1,2,3,4,5};	A is an array initialized to 1..5. Note: A[0]=1.

Arrays:

int R[52];	Save space for 52 integers
int T[2][52];	Save space for two arrays of 52 integers.

note: The PIC18F4626 only has 3692 bytes of RAM, so don't get carried away with arrays.

General C Commands:

Conditional Expressions:

```
!                   not.   !PORTB means the compliment of PORTB.
=                   assignment
==                  test if equal.
```

```
>                  greater than
<                  less than
>=                 greater than or equal
!=                 not equal
```

IF Statement

```
if (condition expression)
{   statement or group of statements
   }
```

example: if PortB pin 0 is 1, then increment port C:

```
if (RB0==1) {
   PORTC += 1;
   }
```

IF - ELSE Statements

```
if (condition expression)
{   statement or group of statements
   }
else {
   alternate statement or group of statements
   }
```

Example: if PortB bit 0 is 1, then increment port C, else decrement port C:

```
if (RB0==1)
   PORTC += 1;
   }
else
   PORTC -= 1;
   }
```

SWITCH (CASE)

```
switch(value)
{
   case value:  statement or group of statements
   case value:  statement or group of statements
   defacult:    statement or group of statements
   }
```

WHILE LOOP

```
while (condition is true) {
   statement or group of statements
   }
```

DO LOOP

```
do {
   statement or group of statements
   } while (condition is true);
```

FOR-NEXT

```
for (starting value; do while true; changes) {
   statement or group of statements
   }
```

Infinite Loop

```
while(1) {
    statement or group of statements
    }
```

note: Zero is false. Anything other than zeros is true. while(130) also works for an infinite loop.

Subroutines in C:

To define a subroutine, you need to

- Declare how this subroutine is called (typically in a .h file)
- Declare what the subroutine is.

The format is

returned_variable_type = subroutine_name(passed_variable_types).

Example: Write a subroutine which returns the square of a number:

```
// Subroutine Declarations

int Square(int Data);

// Subroutines

int Square(int Data) {
   int Result;
   Result = Data * Data;
   return(Result);
   }
```

Standard C Code Structure

So that others can modify your code more easily, a standard structure is to be used. This places all code in the following order:

```
//-----------------------------------
//  Program Name
//
//  Author
//  Date
//  Description
//  Revision History
//----------------------------------

// Global Variables

// Subroutine Declarations
#include <pic.h>        // where PORTB etc. is defined


// Subroutines
void interrupt IntServe(void){}   // holder for interrupts (see week 8)

// Main Routine

void main(void)
{

   TRISA = 0;        // all pins on PORTA are output
   TRISB = 0xFF;     // all pins on PORTB are input
   TRISC = 0;        // all pins on PORTC are output
   ADCON1 = 15;      // PORTA and PORTE are binary (vs analog)
   PORTA = 1;        // initialize PORTA to 1 = b00000001
   PORTC = 3;        // initialize PORTC to 3 = b00000011

   while(1) {
      PORTC = PORTB;     // copy whatever is input to PORTB to PORTC
      };
   }

// end of program
```

Address	Register Name	Bit							
		7	6	5	4	3	2	1	0
0xF80	PORTA	-	-	RA5	RA4	RA3	RA2	RA1	RA0
0xF81	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
0xF82	PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0
0xF83	PORTD	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0
0xF84	PORTE	-	-	-	-	RE3	RE2	RE1	RE0
0xF85	LATA	-	-	LATA5	LATA4	LATA3	LATA2	LATA1	LATA0
0xF86	LATB	LATB7	LATB6	LATB5	LATB4	LATB3	LATB2	LATB1	LATB0
0xF87	LATC	LATC7	LATC6	LATC5	LATC4	LATC3	LATC2	LATC1	LATC0
0xF88	LATD	LATD7	LATD6	LATD5	LATD4	LATD3	LATD2	LATD1	LATD0
0xF89	LATE	-	-	-	-	LATE3	LATE2	LATE1	LATE0
0xF92	TRISA	-	-	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0
0xF93	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0
0xF94	TRISC	TRISC7	TRISC6	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0
0xF95	TRISD	TRISD7	TRISD6	TRISD5	TRISD4	TRISD3	TRISD2	TRISD1	TRISD0
0xF96	TRISE	-	-	-	-	TRISE3	TRISE2	TRISE1	TRISE0
0xF9D	PEIE1	PSPIE	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
0xF9E	PIR1	PSPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
0xF9F	IPR1	PSPIP	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP
0xFA0	PIE2	OSCFIE	CMIE	-	EEIE	BCLIE	HLVDIE	TMR3IE	CCP2IE
0xFA1	PIR2	OSCFIF	CMIF	-	EEIF	BCLIF	HLVDIF	TMR3IF	CCP2IF
0xFA2	IPR2	OSCFIP	CMIP	-	EEIP	BCLIP	HLVDIP	TMR3IP	CCP2IP
0xFAB	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
0xFAC	TXSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D
0xFAD	TXREG	8 bit register (0-255)							
0xFAE	RCREG	8 bit register (0-255)							
0xFAF	SPBRG	8 bit register (0-255)							
0xFB0	SPBRGH	8 bit register (0-255)							
0xFB1	T3CON	T3RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	T3CCP1	TMR3CS	TMR3ON
0xFB2	TMR3	16 bit register (0..65535)							
0xFB4	CMCON	C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0
0xFB5	CVRCON	CVREN	CVROE	CVRR	CVRSS	CVR3	CVR2	CVR1	CVR0
0xFB6	ECCP1AS	ECCPASE	ECCPAS2	ECCPAS1	ECCPAS0	PSSAC1	PSSAC0	PSSBD1	PSSBD0
0xFB7	PWM1CON	PRSEN	PDC6	PDC5	PDC4	PDC3	PDC2	PDC1	PDC0
0xFB8	BAUDCON	ABDOVF	RCIDL	RXDTP	TXCKP	BRG16	–	WUE	ABDEN
0xFBA	CCP2CON	–	–	DC2B1	DC2B0	CCP2M3	CCP2M2	CCP2M1	CCP2M0
0xFBB	CCPR2	16 bit register (0..65535)							
0xFBD	CCP1CON	P1M1	P1M0	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0
0xFBE	CCPR1	16 bit register (0..65535)							
0xFC0	ADCON2	ADFM	–	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0
0xFC1	ADCON1	–	–	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
0xFC2	ADCON0	–	–	CHS3	CHS2	CHS1	CHS0	GODONE	ADON
0xFC3	ADRES	16 bit register (0..65535)							
0xFC5	SSPCON2	GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN
0xFC6	SSPCON1	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
0xFC7	SSPSTAT	SMP	CKE	DA	STOP	START	RW	UA	BF
0xFCA	T2CON	–	T2OUTPS3	T2OUTPS2	T2OUTPS1	T2OUTPS0	TMR2ON	T2CKPS1	T2CKPS0
0xFCB	PR2	8 bit register (0-255)							
0xFCC	TMR2	8 bit register (0-255)							
0xFCD	T1CON	T1RD16	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON

0xFCE	TMR1	16 bit register (0..65535)							
0xFD0	RCON	IPEN	SBOREN	–	RI	TO	PD	POR	BOR
0xFD5	T0CON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
0xFD6	TMR0	16 bit register (0..65535)							
0xFD8	STATUS	–	–	–	NEGATIVE	OV	ZERO	DC	CARRY
0xFF0	INTCON3	INT2IP	INT1IP	–	INT2IE	INT1IE	–	INT2IF	INT1IF
0xFF1	INTCON2	RBPU	INTEDG0	INTEDG1	INTEDG2	–	TMR0IP	–	RBIP
0xFF2	INTCON	GIE	PEIE	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF