MPLAB8 and C Programming:

ECE 376 Embedded Systems

Jake Glower - Lecture #8

Please visit Bison Academy for corresponding
lecture notes, homework sets, and solutions

MPLAB8 and C Programming:

- For step-by-step instructions on how to compile and download a program using MPLAB8 and PICC18, please refer page 3.
- If you're not familiar with C or forgot most of what you learned in ECE 173, don't worry. We'll start with fairly simple C programs and build from there.
- If you want to get an A or B in this course, please do the homework and test it on your PIC board. Writing programs on paper (or copying someone else's code) isn't the same as trying to get it to work in practice. Besides, this course is a lot more fun if you can see your devices actually working.

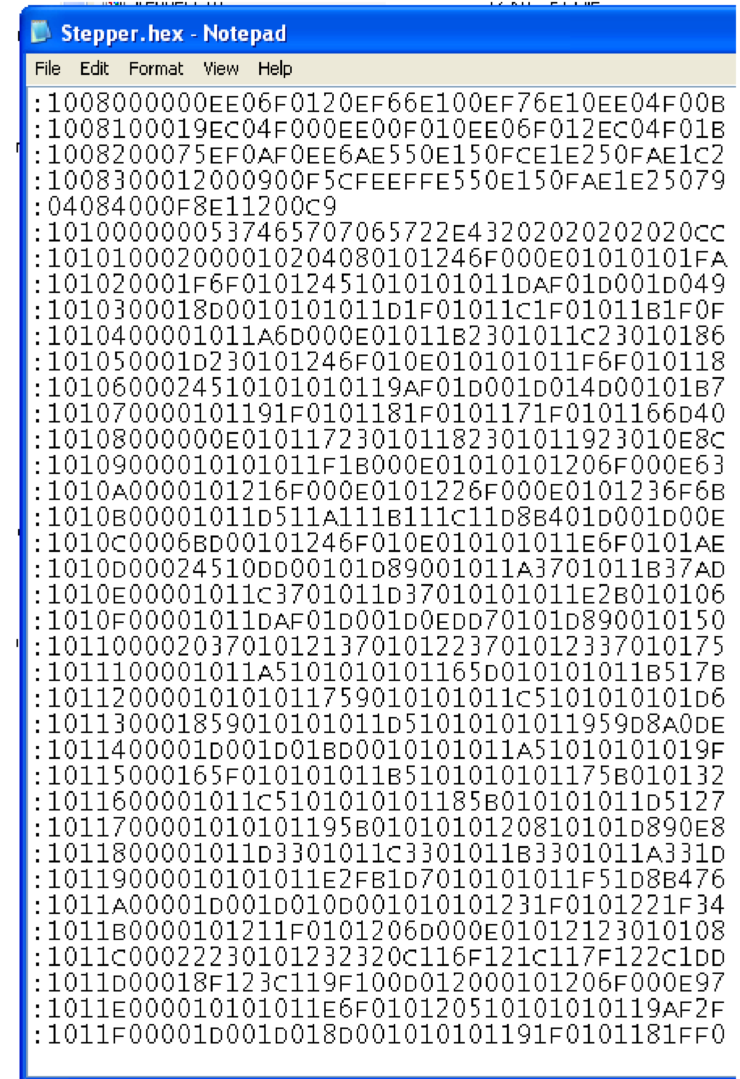
Machine Code

Pre 1950, computers were programmed in machine code.

- Very cryptic
- Hard to understand
- Hard to debug

Example: Stepper Motor Driver

- Machine Code (.hex file)



Assembler

- *Much* superior to machine code
- Semi-meaningful names represent the valid machine operations
- Fast, efficient

Problems:

- Limited instruction set
- Still very cryptic, hard to debug, hard to maintain

Example: Stepper Motor Controller

Stepper.lst - Notepad

File Edit Format View Help

```
;Stepper1.C: 15: void main(void)
;Stepper1.C: 16: {

  _main:                    ; BSR set to: ?

        opt stack 31
        line     19
        movlw    low(0)
        movwf    ((c:3986)),c     ;volatile
        line     20
;Stepper1.C: 20: TRISB = 0;
        movlw    low(0)
        movwf    ((c:3987)),c     ;volatile
        line     21
;Stepper1.C: 21: TRISC = 0;
        movlw    low(0)
        movwf    ((c:3988)),c     ;volatile
        line     22
;Stepper1.C: 22: TRISD = 0;
        movlw    low(0)
        movwf    ((c:3989)),c     ;volatile
        line     23
;Stepper1.C: 23: TRISE = 0;
        movlw    low(0)
        movwf    ((c:3990)),c     ;volatile
        line     24
;Stepper1.C: 24: ADCON1 = 0x0F;
        movlb    1          ; () banked
        movwf    (??_main+2+0)&0ffh
        movlw    low(0Fh)
        movwf    ((c:4033)),c
        movlb    1          ; () banked
        movf     (??_main+2+0)&0ffh,w
        line     26
;Stepper1.C: 26: STEP = 0;
        movlw    low(0)
        movwf    ((c:2)),c
```

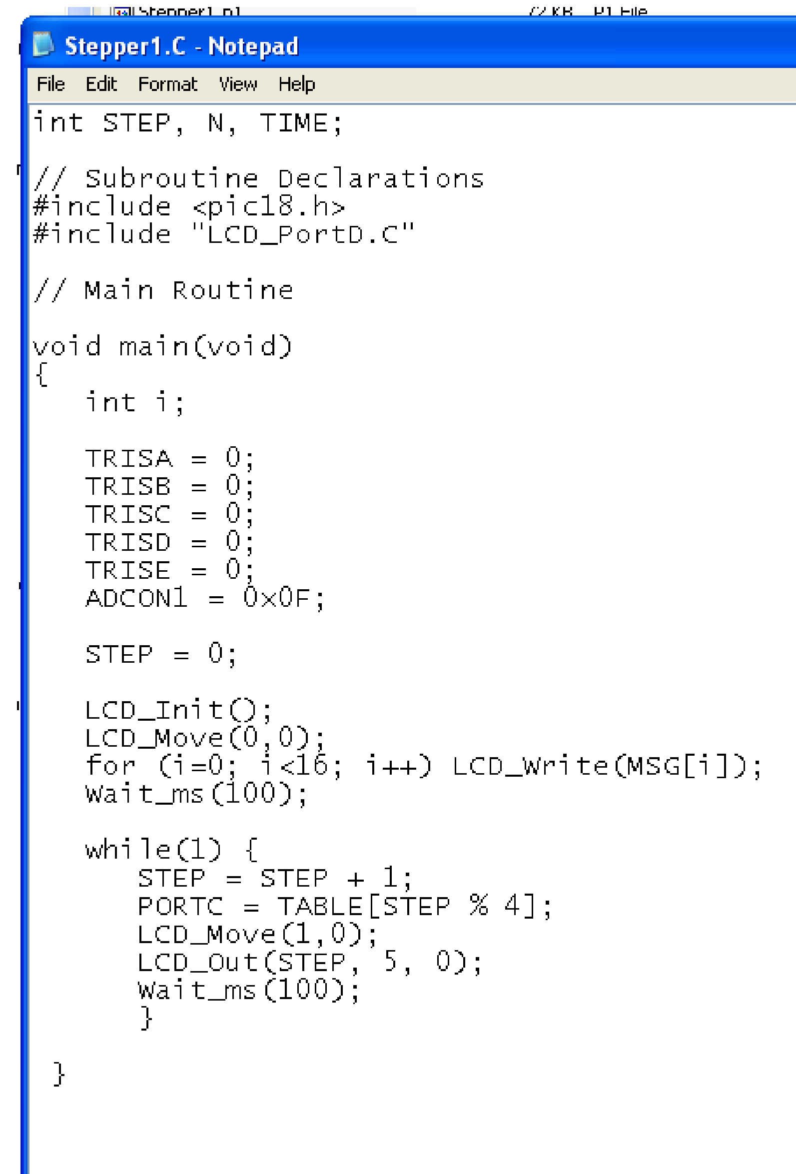
C Language

Adds

- multiply, divide,
- arrays
- for next, do while loops
- if statements

Far easier to write code which is

- Understandable
- Testable
- Reusable



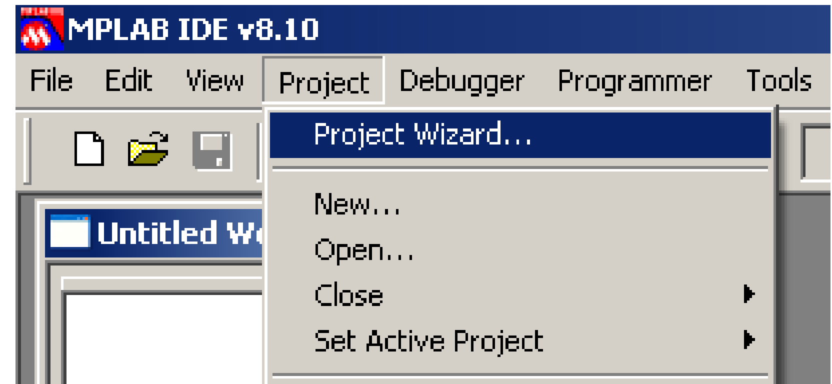

```c
int STEP, N, TIME;

// Subroutine Declarations
#include <pic18.h>
#include "LCD_PortD.C"

// Main Routine

void main(void)
{
   int i;

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

   STEP = 0;

   LCD_Init();
   LCD_Move(0,0);
   for (i=0; i<16; i++) LCD_Write(MSG[i]);
   Wait_ms(100);

   while(1) {
      STEP = STEP + 1;
      PORTC = TABLE[STEP % 4];
      LCD_Move(1,0);
      LCD_Out(STEP, 5, 0);
      Wait_ms(100);
      }

 }
```

Procedure for Compiling a C Program

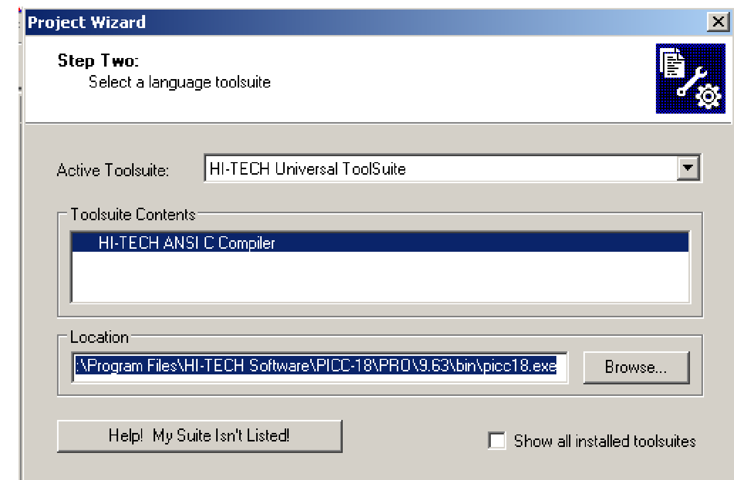
Step 1: Create a directory (if needed)

Step 2: Start MPLAB.

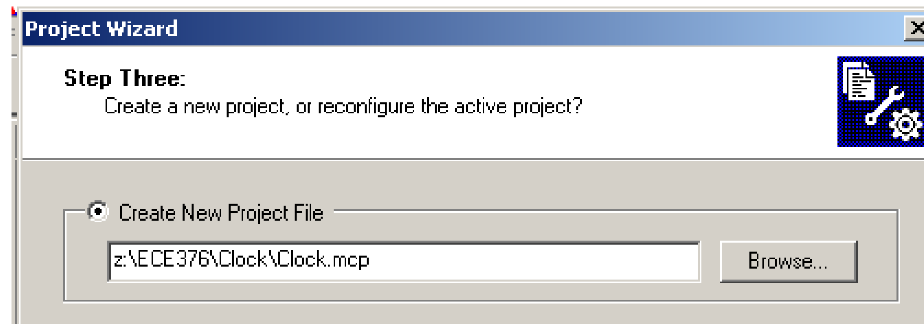
- Go to the program wizard
- Select your device: PIC18F4620
- Select the Hi-Tech C Universal Toolsuite.



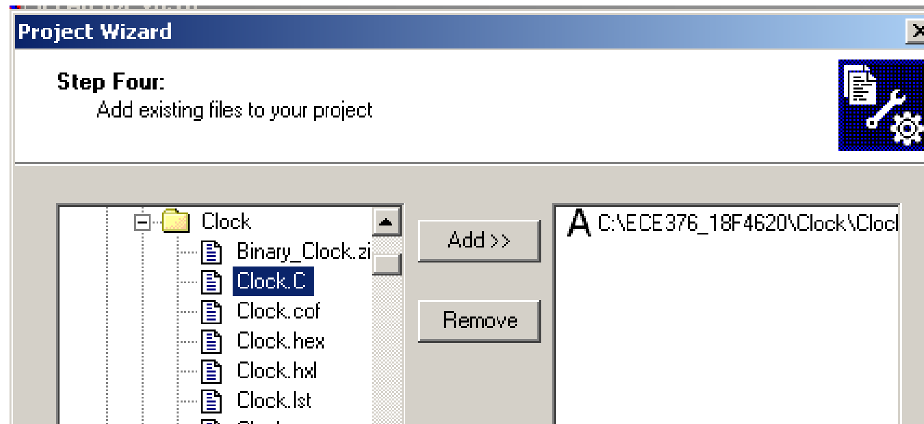
This tells the compiler to interprit your code as C code.



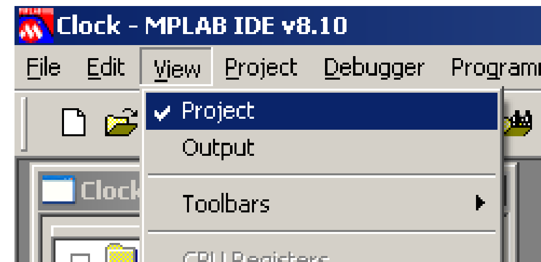
Change the path to your thumb-drive for where the files are located

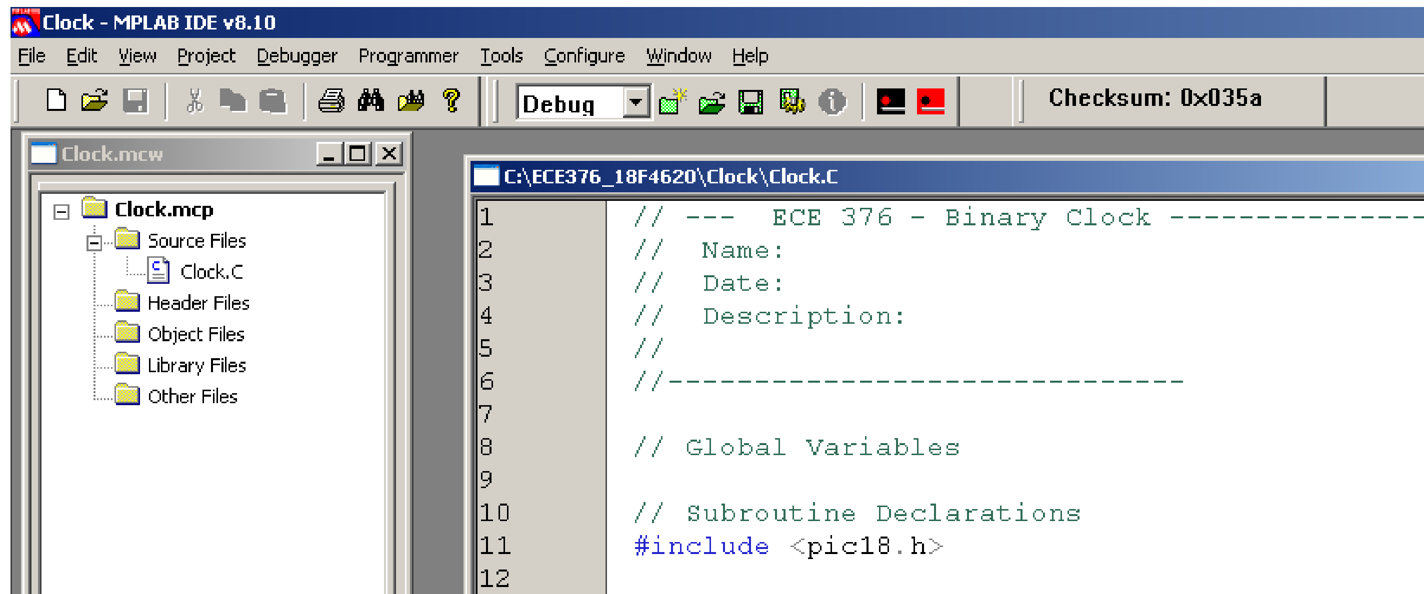


Select the C program you want to compile



View Project

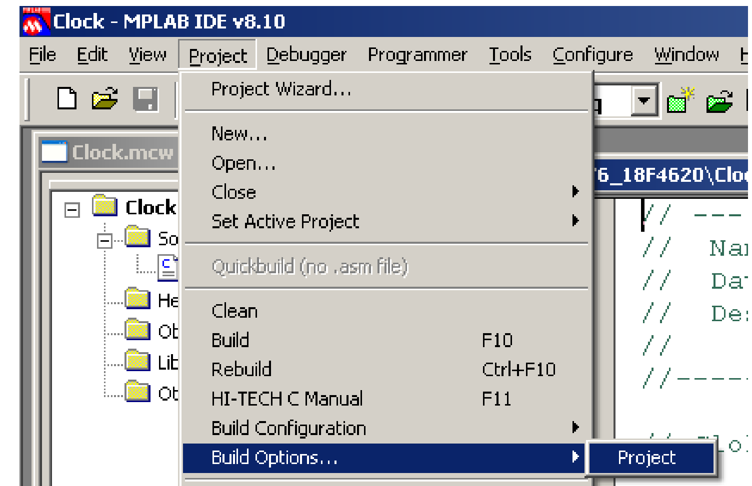




*** important *** Offset your code by 0x800

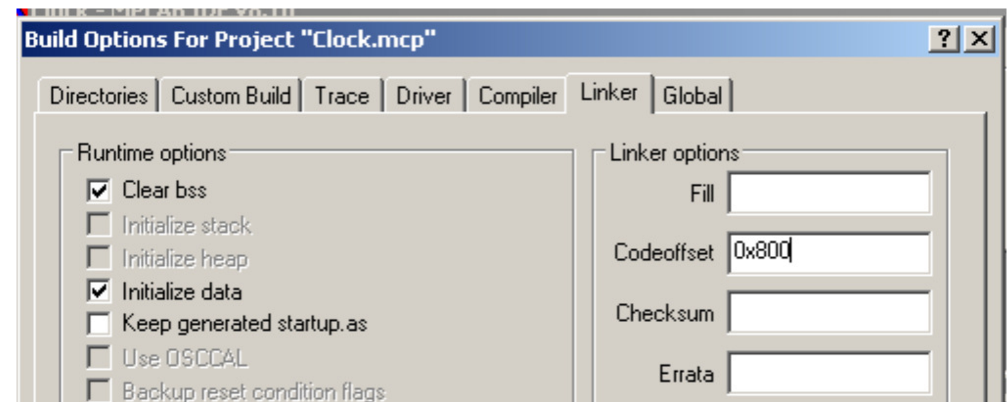
Your code needs to start at 0x800 - after the boot-loader.

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

```
161    153  00FFAC  51FF          movf     (??_main+2+0)&0ffh,w
162    154                        line     29
163    155                     ;Clock.C: 29: PORTA = 0;
164    156  00FFAE  0E00          movlw    low(0)
165    157  00FFB0  6E80          movwf    ((c:3968)),c     ;volatile
166    158                        line     30
167    159                     ;Clock.C: 30: PORTB = 0;
168    160  00FFB2  0E00          movlw    low(0)
169    161  00FFB4  6E81          movwf    ((c:3969)),c     ;volatile
170    162                        line     31
171    163                     ;Clock.C: 31: PORTC = 0;
172    164  00FFB6  0E00          movlw    low(0)
173    165  00FFB8  6E82          movwf    ((c:3970)),c     ;volatile
174    166                        line     32
175    167                     ;Clock.C: 32: PORTD = 0;
176    168  00FFBA  0E00          movlw    low(0)
177    169  00FFBC  6E83          movwf    ((c:3971)),c     ;volatile
178    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.

Forgot C?

If you don't remember C that much, don't worry

- We don't use many of the features of C

Main things we use are...

- if
- if, elseif
- do
- while
- subroutines

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
```

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.

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
    }

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

If ... else ...

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

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

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
    }
```

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.

```
// Subroutine Declarations

int Square(int Data);

// Subroutines

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

Standard C Code Structure

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

// Global Variables

// Subroutine Declarations
#include <pic18.h>        // where PORTB etc. is defined

// Subroutines

// Main Routine

void main(void)
{

   }
```

C vs Assembler

- C compiles into assembler very efficiently
- Claim: 80% efficient
- Actually: C code is 3-10x larger & slower than assembler

Send {1, 2, 3, 4} to PORTA..D

Assembler: 16 instructions

```
      org 0x800

      clrf  TRISA
      clrf  TRISB
      clrf  TRISC
      clrf  TRISD
      clrf  TRISE
      movlw 0x0F
      movwf ADCON1

      movlw 1
      movwf PORTA
      movlw 2
      movwf PORTB
      movlw 3
      movwf PORTC
      movlw 4
      movwf PORTD
  Loop:
      goto  Loop
```

C: 29 assembler instructions

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

// Main Routine

void main(void)
{

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

   PORTA = 1;
   PORTB = 2;
   PORTC = 3;
   PORTD = 4;

   while(1);
   }
```

Compilation Results:

- Each instruction takes 16 bits (2 bytes / instruction
- 16 instructions when written in assembler
- 58 bytes = 29 instructions when written in C
- 1.81 x larger

```
Memory Summary:
    Program space        used   3Ah (    58) 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%)
```

Example 2: 32-Bit Counter

Assembler: 15 instructions

```
      org 0x800
      clrf  TRISA
      clrf  TRISB
      clrf  TRISC
      clrf  TRISD
      clrf  TRISE
      movlw 0x0F
      movwf ADCON1
Loop:
      incfsz PORTD,F
      goto   Loop
      incfsz PORTC,F
      goto   Loop
      incfsz PORTB,F
      goto   Loop
      incfsz PORTA,F
      goto   Loop
```

C: 67 assembler instructions

```
#include <pic18.h>
void main(void)
{
   unsigned long int X;

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

   X = 0;

   while(1) {
      X = X + 1;
      PORTD = X;
      PORTC = X >> 8;
      PORTB = X >> 16;
      PORTA = X >> 32;
      }
   }
```

Compilation results are:

```
Memory Summary:
    Program space          used    86h (   134) of 10000h bytes   (  0.2%)
    Data space             used     5h (     5) 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 code compiles into 67 lines of assembler (134/2).

In-Line Assembler:

- Almost all C compilers allow you to include assembler code
- Makes it easy on the compiler: it doesn't have to do anything

Normally you don't want to do this:

- assembler is much harder to understand and debug
- assembler is much harder to maintain

Single Instruction

```
asm("   nop");
```

Multiple Instructions

```
#asm
   nop
   nop
   nop
#endasm
```

How Long does C Code Take to Execute?

Assembler is easy to determine execution time

- Count number of instructions
- Each instruction is one clock (2 for jumps)

C-code is hard to determine

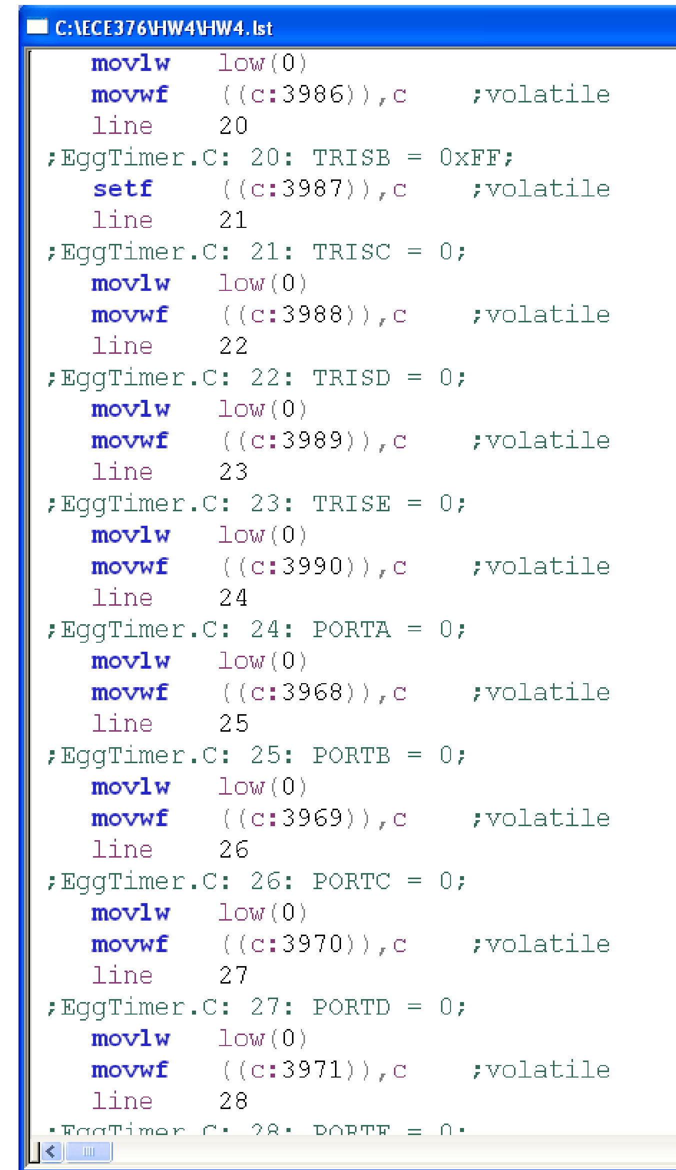
- Each line of C compiles into 1 or more lines of assembler
- Different C compilers result in different answers

Option 1:

- Look at the .lst file and count number of assembler instructions

Option 2:

- Toggle a pin and determine the time experimentally



Execution Time: Counting Mod 64

C-Code

```
unsigned char i
while(1) {
        i = (i + 1)% 64;
        if(i == 0) PORTC += 1;
        }
```

Measure the freuqncy on RC0

- f = 4890.0Hz

$$N_{64} = \left(\frac{10{,}000{,}000}{2 \cdot 4890}\right) = 1022.49$$

$$N_1 = \frac{1022.49}{64} = 15.98 \approx 16$$

It takes 16 clocks to count mod 64

Execution Time: Counting Mod 63

C-Code

```
unsigned char i
while(1) {
      i = (i + 1)% 63;
      if(i == 0) PORTC += 1;
      }
```

Measure the frequency on RC0:

- f = 140.6 Hz

$$N_{63} = \left(\frac{10{,}000{,}000}{2 \cdot 140.6} \right) = 35,561.88$$

$$N_1 = \left(\frac{35{,}561.88}{63} \right) = 564.47$$

It takes 564 (ish) clocks to count mod 63

Execution Time: Integer Multiply

C-Code

```
unsigned int A, B, C;
unsigned char i;
   A = 1234;
   B = 5678;
   while(1) {
      i = (i + 1)% 64;
      if(i == 0) PORTC += 1;
      C = A*B;
      }
```

Measure the frequency on RC0:

- f = 192,2Hz

$$N_{64} = \left(\frac{10{,}000{,}000}{2 \cdot 192.2}\right) = 26,014.57$$

$$N_1 = \left(\frac{26.014.57}{64}\right) = 406.48$$

- 16 clocks to count mod 64 plus 390 clocks to do an integer multiply

It takes 390 (ish) clocks to do an integer multiply

Execution Time: Floating Point Multiply

C-Code:

```
float A, B;
A = 1.00001;
B = 0.02;
while(1) {
   i = (i + 1)% 64;
   if(i == 0) PORTC += 1;
   B = B * A;
   }
```

RC0 = 66.1Hz

$$N_{64} = \left(\frac{10{,}000{,}000}{2 \cdot 66.1}\right) = 75{,}642.97$$

$$N_1 = \left(\frac{75{,}642.97}{64}\right) = 1181.92$$

Each loop takes about 1182 clocks.

- 16 clocks to count mod 64 plus 1166 clocks to do a floating point multiply

It takes 1166 clocks to do a floating point multiply

Summary:

C code is a *lot* easier to write than assembler

C code is a *lot* easier to understand than assembler

Sometimes, it's easiest to use experimental methods to determine something

Example: Execution Time

- Toggle a pin each time you go through your main routine
  - *Tie the pin to a speaker*
  - *Measure the frequency of the resulting square wave*
- Set a pin prior to a funciton call, clear it when you return
  - *Measure the pulse width with an oscilloscope*
  - *The pulse-width is the execution time*

Variable Names (#include PIC18.h)

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