MPLAB8 and C:

- 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.

Background

Back in the 1960's, computers were programmed in machine code. The operator would set switches according to the binary code corresponding to each line of code, push a button, and set the switches for the next line of code.

Machine code is very cryptic. A program for a PIC which counts on PORTC looks like the following:

```
060000000A128A11F92F1B
0E0FF200831603138701831287018 70AFE2FDF
00000001FF
```

Assembler is *much* superior to machine code. Semi-meaningful names represent the valid machine operations, as described in the previous notes. The previous code would look like the following

```
                    _main
                    clrf      TRISC
                    clrf      PORTC
_loop               incf      PORTC,F
                    goto      _loop
```

This is a lot easier to understand than the machine code. It is still very cryptic, however. In addition, assembler has a limited set of commands. The PIC we're using, for example, can

- Add, Subtract
- Load, Store
- Shift left, shift right, and
- Do boolean operations.

Using these limited instructions, you can do anything, such as implement a Fourier transform. The algorithm will be very cryptic, however.

C is a high-level assembler which has some useful functions, such as

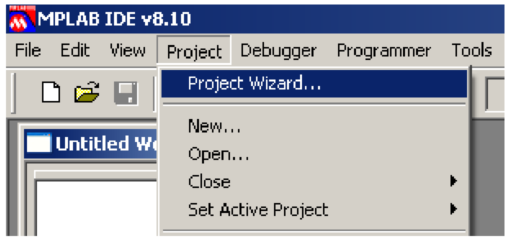
- multiply, divide,
- arrays
- for next, do while loops
- if statements

Procedure for Compiling a C Program

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

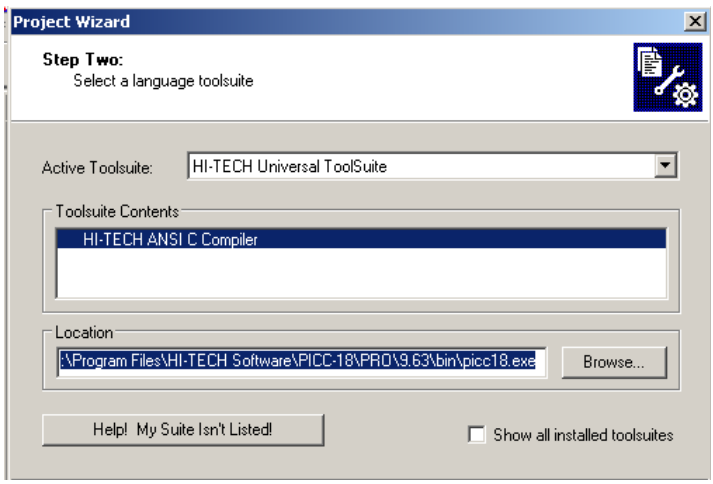
z:\ECE376\Clock

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



Select your device: PIC18F4620

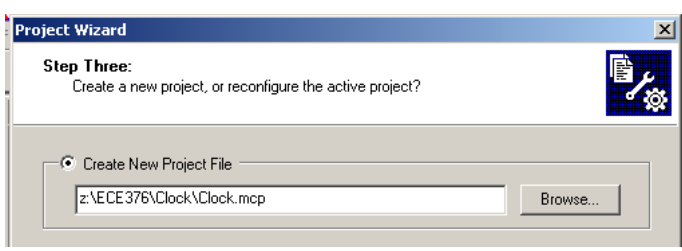
Select the Hi-Tech C Universal Toolsuite.



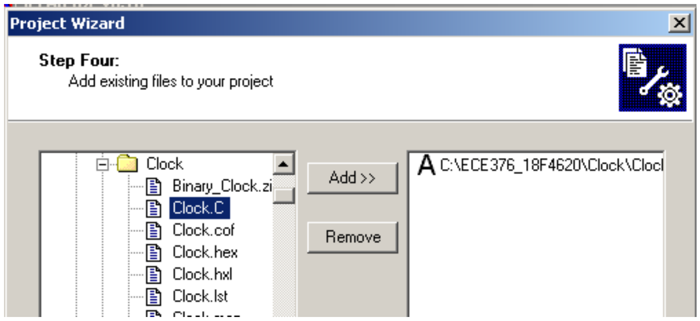
This tells the compiler to interprit your code as C code. Note that if this isn't an option under the Active Toolsuite, there's a problem. This usually means the C compiler is in a read-only directory and needs the permissions changed by a system administrator.

Assuming that works...

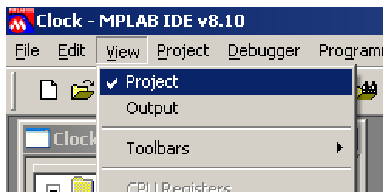
Change the path to your z-drive for where the files are located



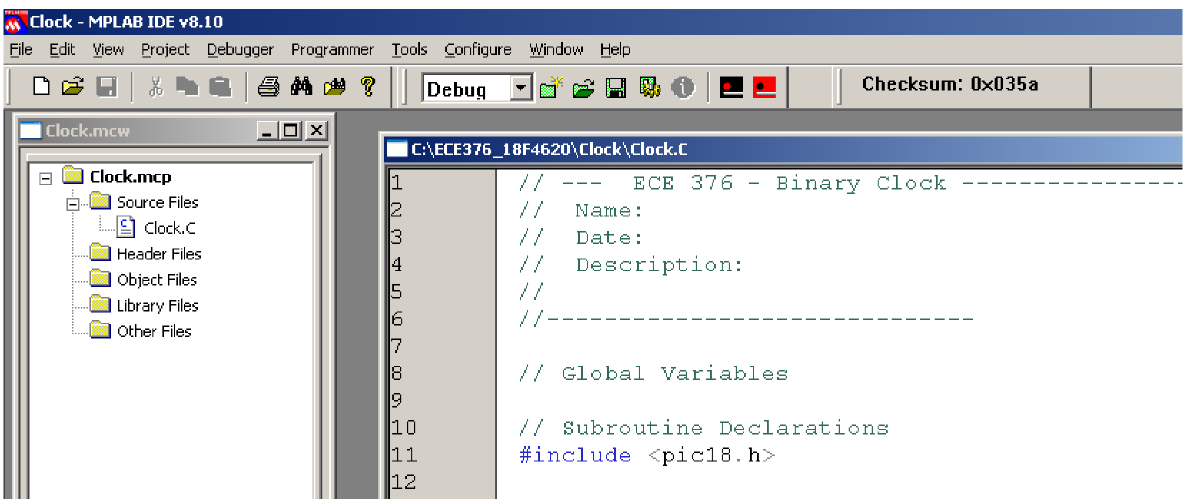
Select the C program you want to compile (usually the name of the zip file



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



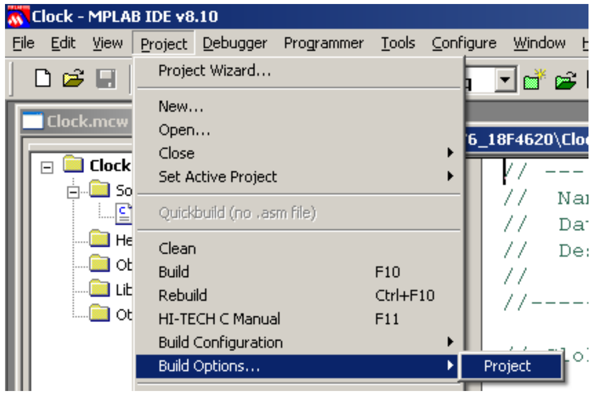
You should get the following screen:



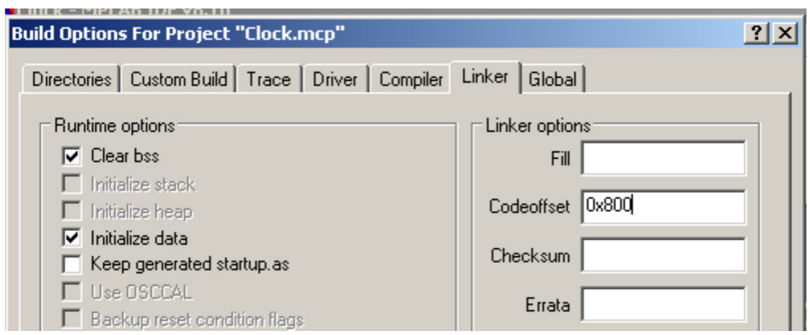
*** important *** Offset your code by 0x800

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.

If you don't remember C that much, don't worry: we don't use many of the features of C. I personally treat C like assembler - only with a multiply command. Another theme you'll see is you can do just about anything with an IF statement. The code may not be the most efficient - but as long as it's understandable and works, it's usually good enough. If you really want efficiency, use assembler.

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

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

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
   default:     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
   TRISD = 0;        // all pins on PORTD are output
   TRISE = 0;        // all pins on PORTE 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) {
      PORTD = PORTB;    // copy whatever is input to PORTB to PORTD
      };
   }

// end of program
```

C vs Assembler

Assembler is very powerful. In assembler, you have complete control over where your code compiles and where your variables are stored. Assembler is also very fast and efficient. Assembler is very difficult to read, however, and tends to be throwaway code. It's easier to start from scratch and ignore already written assembler code and start over than to modify existing code.

C is a little constraining. However, it is easier to write C code which is understandable and can be reused with relative ease.

The following examples of C code and its assembler counterpart

1. Binary Outputs

Write a program which sends the numbers {1, 2, 3, 4} to PORTA..D

C-Code

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

// Subroutines

// 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:

```
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%)
```

This C code compiles into 29 lines of assembler (58 bytes: each instruction is 16 bits or two bytes)

- **Assembler: 16 instructions**
- **C Code: 29 instructions (81% larger)**

Example 2: 32-Bit Counter

One nice feature of C is that you have more than just 8-bit variables. You can also use

- 8-bit variables (char)
- 16-bit variables (integer)
- 32-bit variables (long integer)
- 32-bit floating point numbers (float)
- 64-bit floating point numbers (double)

When you copy a 32-bit variable into an 8-bit variable

- The low 8-bits are copied
- The other bits are ignored

Suppose you want to count using a 32-bit variable. This code could be

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

// Subroutines

// Main Routine

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;
      }
   }
```

Note: The command X >> 8 shifts X right eight times. This results in

- Bits 0..7 going to PORTD
- Bits 8..15 going to PORTC
- Bits 16..23 going to PORTB, and
- Bits 24..31 going to PORTA

The 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).

Now, the conversion to assembler isn't so easy to follow. The single line fo C code

```
X = X + 1
```

compiles into 25 lines of assembler as follows:

```
;Test.C: 24: X = X + 1;
   movlw    01h
   movlb    0    ; () banked
   movlb    0    ; () banked
   addwf    ((main@X))&0ffh,w
   movlb    0    ; () banked
   movlb    0    ; () banked
   movwf    ((main@X))&0ffh
   movlw    0
   movlb    0    ; () banked
   movlb    0    ; () banked
   addwfc   ((main@X+1))&0ffh,w
   movlb    0    ; () banked
   movwf    1+((main@X))&0ffh
   movlw    0
   movlb    0    ; () banked
   movlb    0    ; () banked
   addwfc   ((main@X+2))&0ffh,w
   movlb    0    ; () banked
   movwf    2+((main@X))&0ffh
   movlw    0
   movlb    0    ; () banked
   movlb    0    ; () banked
   addwfc   ((main@X+3))&0ffh,w
   movlb    0    ; () banked
   movwf    3+((main@X))&0ffh
   line     25
```

The compiler can be pretty tricky as well. Shifting the data to PORTA..D is as follows:

```
;Test.C: 25: PORTD = X;
   movff    (main@X),(c:3971)    ;volatile
   line     26
;Test.C: 26: PORTC = X >> 8;
   movff    0+1+(main@X),(c:3970)    ;volatile
   line     27
;Test.C: 27: PORTB = X >> 16;
   movff    0+2+(main@X),(c:3969)    ;volatile
   line     28
;Test.C: 28: PORTA = X >> 32;
   movff    (main@X),(c:3968)    ;volatile
   line     29
```

X is a 32-bit variables, stored in four bytes in RAM. The C compiler places the low byte at memory location main@x. The next byte is at main@x+1, and so on.

The net result is

Result:

- **Assembler: 21 lines of code**
- **C: Compiles into 67 lines of assembler (3.19x larger)**

In-Line Assembler:

In almost every C language, there is a way to insert assembler code into your C code. With Hi-Tech C, the commands are

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

This inserts the assembler command "nop" into the code at this point in your C code.

You can also insert several lines of assembler with the compiler directives #asm and #endasm

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

Normally you don't want to do this:

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

but

- assembler is 3-10 times smaller and faster than C

The times you would do this are

- When your compiled C code doesn't fit in your processor. You need to make the code smaller somehow.
- When your compiled C code takes too long to execute. You need to speed it up somehow.

For the latter, what's often done is monitor how much time is spend in each routine. The routine which is eating up most of your time is then hand-coded into assembler (biggest bang for the buck). This results in code which is much harder to maintain, but it is faster.

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