EE2A2

Introductory Material: Using the C language on a PIC Processor

1.0 Aims

(i) To take a bare microprocessor and to convert it into a working micro-controller system programmed using the C language.

(ii) To use IO-mapped data structures to talk to interface pins.

(iii) To consider the basic functionality of a synchronous finite state machine and the equivalent implementation on a processor.

(iv) To combine multiple, disjoint program fragments together into one package that can be rapidly demonstrated.

(v) To implement logical bit-wise operations in software

(vi) To implement look-up tables on both inputs and outputs (lambda logic) of synthesised finite state machines.

This experiment is designed to allow students to familiarise themselves with developing a microcontroller system programmed with the C language. It will include programming the device using an in-circuit programmer, developing IO-mapped structures, the use of bit-wise logic operators and the relationships between a microcontroller and a synchronous finite state machine.

2.0 Learning Outcomes

Deliverables:

i) A working system.

ii) Well-structured and -commented PIC programs that interface to the outside world according to the specification given in Section 3.1.

iii) A log book containing any preparatory work undertaken outside the lab, design decisions, test procedures and a discussion of the issues involved.

iv) Ability to generate your own mark scheme and rigorous evaluation procedures (can you pre-empt a cynical boss?).

What will be assessed:

i) The program source code, with particular emphasis on adequate comments, program layout and elegance.

ii) A practical demonstration.

iii) The adequacy of the log book (lab books not written up during the experiment will incur penalty marks).

3.0 Activities

3.1 Specification

3.1.0 Demonstrate that clock microprocessor is running at 4 MHz

The PIC 18F27K40 may be clocked at speeds from DC to 64 MHz. A high-precision internal oscillator block (HFINTOSC) may be used to select frequencies range up to 64 MHz. Examine the include file (18F27K40.h) to find out which parameters and commands may be used. A clock signal at a frequency of one quarter of the main frequency may be configured to appear on pin RA.6 (Hint: #use delay(internal=64MHZ,clock_out)).

3.1.1 Demonstrate that clock microprocessor is running at 8 MHz

Repeat 3.1.0 with the main clock running at 8 MHz Ideally, find out how to change the clock speed once the main program has started. Even better, read Section 4 of the data sheet to see how manufacturer's define their products and look at the 'lst' file to see how your code is converted into register settings.

3.1.2 Demonstrate that clock microprocessor is running at 16 MHz

Repeat 3.1.0 with the main clock running at 16 MHz.

3.1.3 Single Pin Output

A synchronous finite state machine is to be simulated where the output connected to Port C0 toggles on each clock cycle (the equivalent of a T-type flip-flop with the T input connected high). You are to simulate a 2 Hz clock applied to the synchronous finite state machine, such that the output changes at a 1 Hz rate. A single-bit memory-mapped IO structure, or variable, should be used. Ensure that the output rate remains constant as the clock frequency is varied repeatedly (4 MHz, 8 MHz, 16 MHz)

3.1.4 Multiple Pin Output (i)

A synchronous finite state machine is to be simulated where the output connected to Port C0..5 implements a 6-bit pure-binary up-counter. You are to simulate a 1 Hz clock applied to the synchronous finite state machine. A six-bit memory-mapped IO structure, or variable, should be used.

3.1.5 Multiple Pin Output (ii)

A synchronous finite state machine is to be simulated where the output connected to Port C0..5 implements a 6-bit pure-binary down-counter. You are to simulate a 1 Hz clock applied to the synchronous finite state machine. A six-bit memory-mapped IO structure, or variable, should be used.

3.1.6 Multiple Pin Output with output logic

A synchronous finite state machine is to be simulated where the output connected to Port C0..5 implements a 6-bit Gray-code up-counter. This is to be achieved by using the equivalent of 'output logic' to convert between a binary value and a Gray code value – this is called a look-up or hashing table. You are to simulate a 1 Hz clock applied to the synchronous finite state machine. A six-bit memory-mapped IO structure, or variable, should be used.

3.1.7 Multiple Output Groups with output logic

A synchronous finite state machine is to be simulated where the output connected to Port C0..2 implements a 3-bit pure-binary up-counter, whilst Port C3..5 implements a 3-bit Gray-code up-counter. You are to simulate a 1 Hz clock applied to the synchronous finite state machine. A suitable memory-mapped IO structure should be used. Note, multiple structures mapped to the same IO locations will be required.

3.1.8 Multiple Input Groups – demonstration of bit-wise AND operation

Connect the 8-way DIP switch to the pins of port B0..3 and port A0..3. Connect the other side of the switches to 0V. Configure all pins of Port to be inputs with weak pull-up resistors (read section 15 of the PIC 18F27K40 manual, the CCS compiler manual and the header file to understand about the WPU register). Configure the IO mapping structure such that a two-bit structure-member variable is mapped to pins B0..1 and a second two-bit structure-member variable is mapped to pins B2..3. Bit-wise AND the two two-bit variables together an output the result to a structure-member variable mapped to pins C0..1.

3.1.9 Multiple Input Groups – demonstration of bit-wise OR operation

Repeat 3.1.8 with the AND operation replaced by an OR operation.

3.1.10 Multiple Input Groups – demonstration of bit-wise NOT operation

Repeat 3.1.8 with the pins C0..3 being the inverse (NOT) of pins B0..3.

3.1.11 Multiple Input Groups – demonstration of bit-wise Exclusive-OR operation

Repeat 3.1.8 with the AND operation replaced by an Exclusive-OR operation.

3.1.12 Demonstration of Set-Reset flip-flop simulation

Assume that B0 is the set input and that B1 is the reset input of a set-reset flip flop. Pin C0 should be assumed to be the Q output – implement the bistable element.

3.1.13 Demonstration of input look-up tables

Use the four inputs derived from pins B0..3 to drive the indices of an input look-up table. Connect eight outputs to pins C0..7 and implement either a running-light or bar-graph type display. The input pins will select one of 16 non-linearly spaced sampling rates. Make sure that your readership understands why input look-up tables are so good at covering all possible input conditions in a single line of code.

3.1.14 Demonstration of ‘bright-eyed and bushy tailed capability’

Be creative and play – demonstrate something interesting (you have actually been learning about inputs, outputs and mapping multiple structures to the same IO pins)

3.1.15 Combined all above experiments into one program

Use pins A0..3 to drive a ‘case’ statement (or sixteen ‘if’ statements) to combine all the sub-sections into a single program that will allow you to demonstrate all the individual component programs extremely rapidly.

That’s it – now do it!

4.1 Preparatory Work

The following sections aim to guide you through some of the design processes.

4.1.1 Hardware

The PIC 18F27K40 is ideal for this laboratory experiment as it contains an internal RC oscillator (thus saving two pins, a crystal and two capacitors). Pins '*Vss*' are connected to 0V and pin '*Vdd*' is connected to +5V (Decouple this supply with a 100 nF capacitor as close to the IC pins as possible). Overall, there are 28 pins on the device of which probably 20 may be freely used during your experiments, see Figure 1.

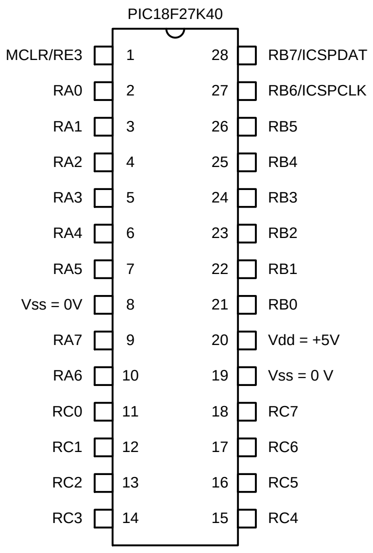


Figure 1: PIC 18F27K40 microcontroller

You will use the PICKit 3 to program the device. This contains a 6-way socket of which pins 1 to 5 will be used – Pin 1 is identified by a triangle. The interconnections are shown in Figure 2 and the PICKit 3 requires power derived from an external source (pins 2 and 3).

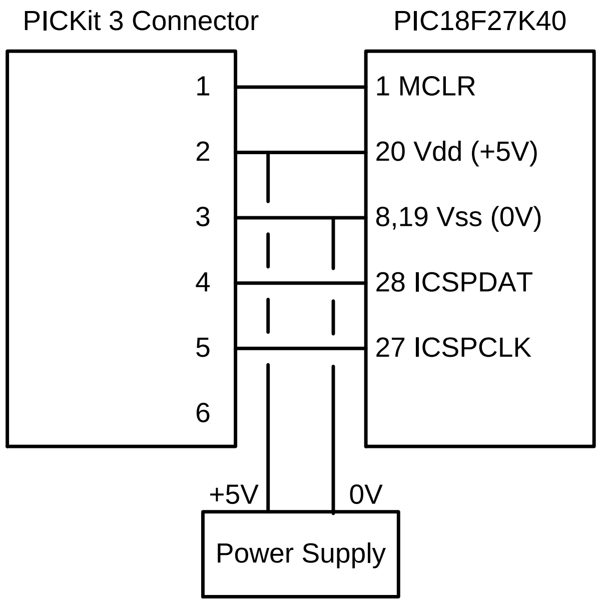


Figure 2: In Circuit Serial Programming Interface

It is suggested that you connect the 8 single-pole, single-throw (SPST) switch module to pins RB0..3 and RA0..3. This choice avoids the two programming pins on RB6 and RB7 and allows the programmer to be left permanently connected to your circuit. The other side of these switches should be connected to 0V, such that when the switch is closed a logic '0' is applied to the relevant

IO pin and when the switch is open a logic '1' is applied as a result of the internal pull-up resistors within the PIC. Eight LEDs should be connected to pins on Port C0..7, each being protected by a series 270R series resistor. Connect pins RB6, RB7 and RE3 to the PICkit 3 programming interface.

PW1. Draw a complete schematic circuit diagram of the PIC 18F27K40, programmer and the interface components. This will be the circuit diagram that you will wire up within the laboratory. All power supply connections, LEDs, resistors and switches should be clearly labelled.

5.1 Laboratory Activities

Starting with a bare microcontroller integrated circuit is going to be scary for the first time as there are so many things to go wrong. The development flow chain is shown in Figure 3.

First, we enter the C code using a text editor. I prefer to use the embedded editor within the CCS Integrated Development Environment (IDE) as this provides the most rapid code development cycle. Other exceptionally good editors that give you assistance as you type include Notepad++.

The flow chain now needs a compiler to generate machine executable code. This compiler will be specific to the device you are integrating and will be very different in nature to a standard PC compiler. Examples include the CCS PCWH which will cost you $500 per machine.

Having generated the executable code, this must be transferred to a suitable programming software environment. To keep things minimally simple, I tend to use the MPLAB Integrated Programming Environment. For these laboratory sessions, you will connect a PICKIT 3 programmer via a USB interface to your PC.

Finally, the programmer will transfer data to your microcontroller using a serial interface sometimes known as an In Circuit Serial Programmer (ICSP).

More details about each of these software suits is provided in Appendix 1.

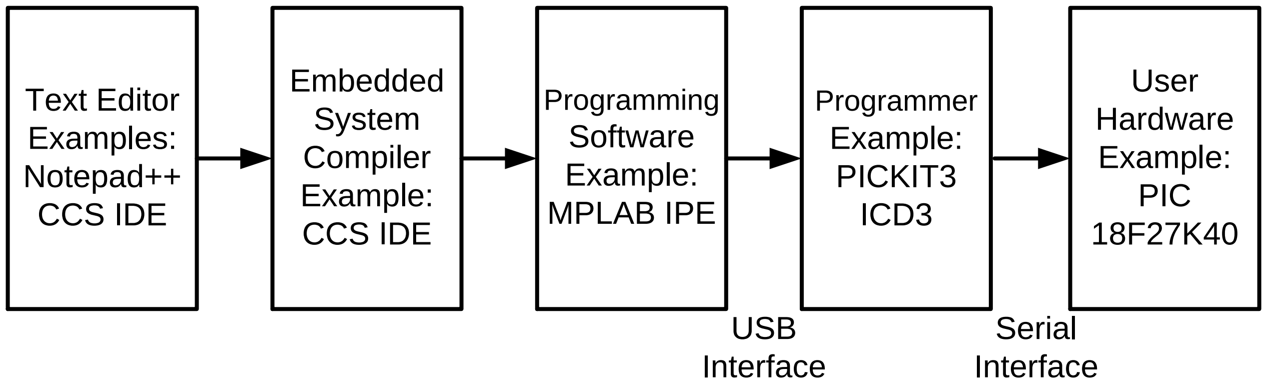

Text Editor Examples: Notepad++ CCS IDE
Embedded System Compiler Example: CCS IDE
Programming Software Example: MPLAB IPE
Programmer Example: PICKIT3 ICD3
User Hardware Example: PIC 18F27K40
USB Interface
Serial Interface

Figure 3: Development flow chain

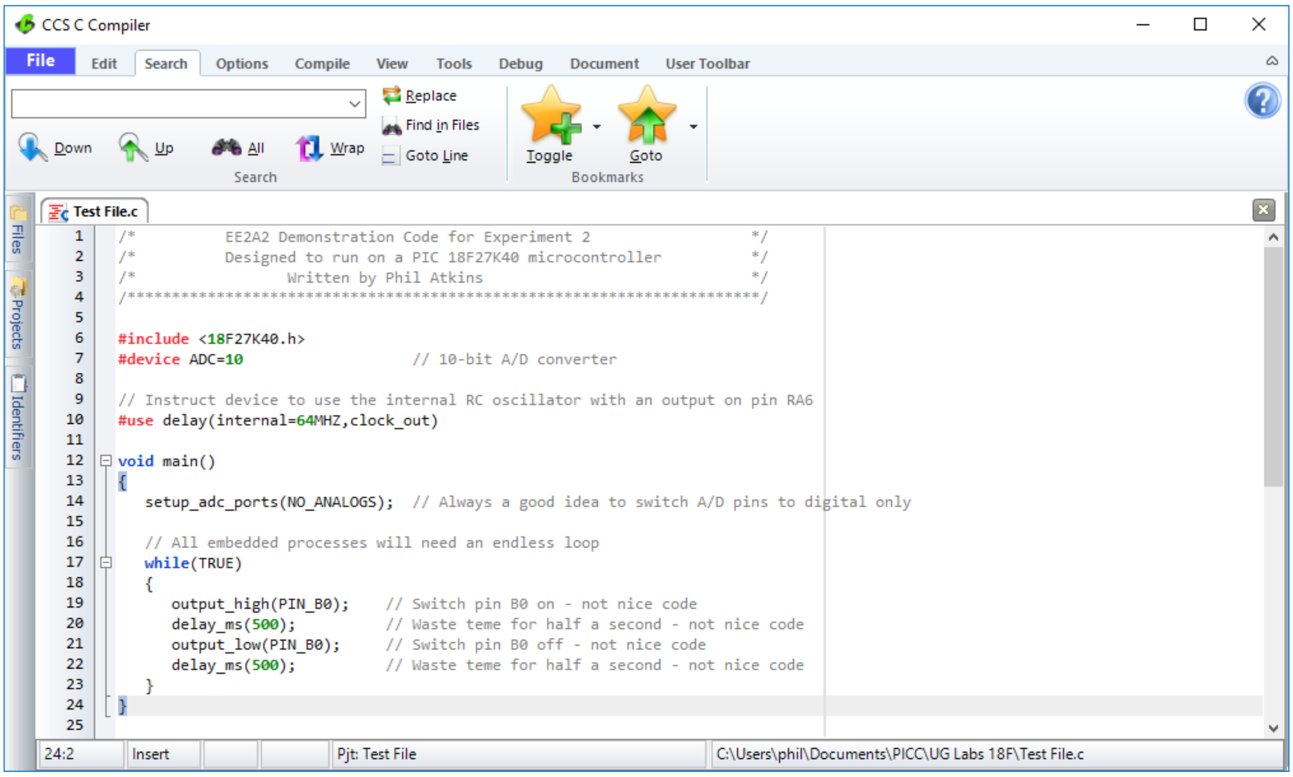
Appendix 1

Development Flow Chain

CCS Environment

Type ‘PIC C Compiler’ in the Windows program search box. It is likely that the compiler will open with the ‘Search’ tab in focus. Click the ‘File’ tab, then the ‘New’ item pull-down selection and finally click on ‘Source File’. A ‘Save As’ dialogue box will appear, navigate to you Y drive files and enter a suitable filename such as ‘EE2A Lab 2.c’.

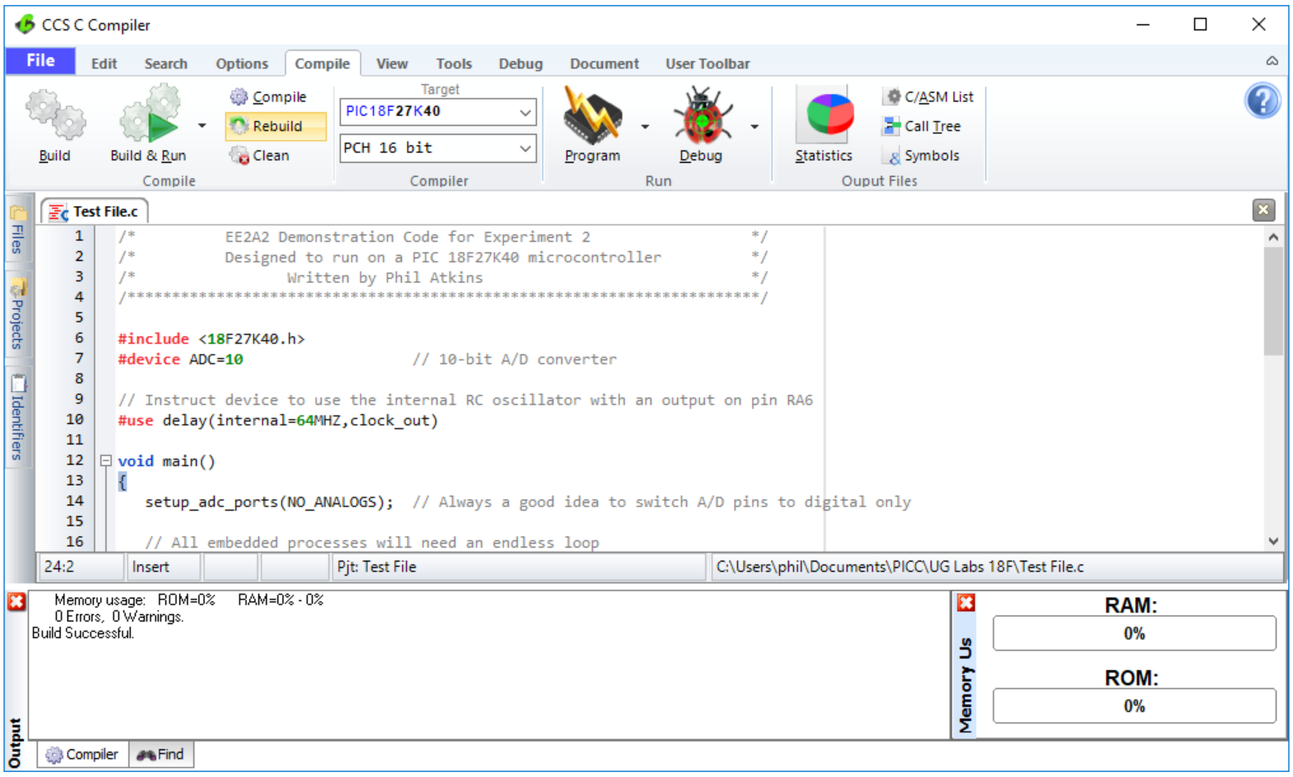
You can now enter a few lines of test code. Most experienced designers enter a very small number of lines between compiles (typically five new lines) and then check that they function as expected. You may get disheartened trying to fix bugs in large programs entered in one go.



```
/*          EE2A2 Demonstration Code for Experiment 2                     */
/*          Designed to run on a PIC 18F27K40 microcontroller             */
/*                   Written by Phil Atkins                               */
/*************************************************************************/

#include <18F27K40.h>
#device ADC=10                       // 10-bit A/D converter

// Instruct device to use the internal RC oscillator with an output on pin RA6
#use delay(internal=64MHZ,clock_out)

void main()
{
   setup_adc_ports(NO_ANALOGS);  // Always a good idea to switch A/D pins to digital only

   // All embedded processes will need an endless loop
   while(TRUE)
   {
      output_high(PIN_B0);      // Switch pin B0 on - not nice code
      delay_ms(500);            // Waste teme for half a second - not nice code
      output_low(PIN_B0);       // Switch pin B0 off - not nice code
      delay_ms(500);            // Waste teme for half a second - not nice code
   }
}
```

24:2 | Insert | Pjt: Test File | C:\Users\phil\Documents\PICC\UG Labs 18F\Test File.c

Having entered the code, save it and click the ‘Compile’ tab. Now click ‘Build’. A pop-up menu will appear to inform you of success, or failure. After a few seconds the screen should revert to something like:



Target: PIC18F27K40
PCH 16 bit

```
/*          EE2A2 Demonstration Code for Experiment 2              */
/*          Designed to run on a PIC 18F27K40 microcontroller      */
/*                   Written by Phil Atkins                        */
/***********************************************************************/

#include <18F27K40.h>
#device ADC=10                      // 10-bit A/D converter

// Instruct device to use the internal RC oscillator with an output on pin RA6
#use delay(internal=64MHZ,clock_out)

void main()
{
   setup_adc_ports(NO_ANALOGS);  // Always a good idea to switch A/D pins to digital only

   // All embedded processes will need an endless loop
```

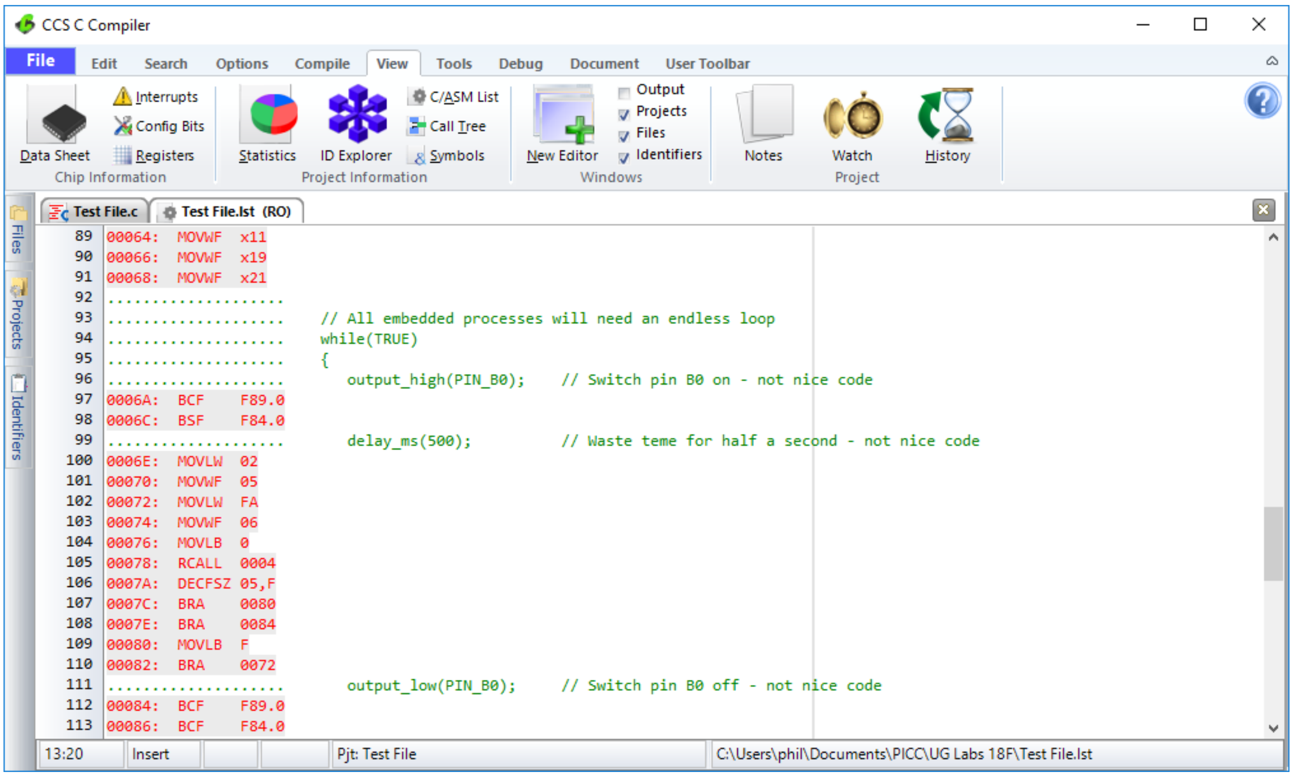
```
24:2    Insert    Pjt: Test File    C:\Users\phil\Documents\PICC\UG Labs 18F\Test File.c
```

```
   Memory usage:   ROM=0%     RAM=0% - 0%
   0 Errors,  0 Warnings.
Build Successful.
```

RAM:
0%
ROM:
0%

If there are errors, hopefully some information is provided about the type and location of the source of the error.   However, an error such as a missing semicolon at the end of a line may have occurred many lines before.

You have now generated some operational code, experienced programmers will always keep an eye on the assembly code generated.   Do this by clicking the ‘View’ tab and the ‘C\ASM List’ icon. You will be able to scroll down through your original C code statements and the assembly code that has been generated to implement your code – compilers do make mistakes and you will be expected to find them in the workplace.



```
00064:  MOVWF  x11
00066:  MOVWF  x19
00068:  MOVWF  x21
....................
....................      // All embedded processes will need an endless loop
....................      while(TRUE)
....................      {
....................         output_high(PIN_B0);    // Switch pin B0 on - not nice code
0006A:  BCF    F89.0
0006C:  BSF    F84.0
....................
....................         delay_ms(500);          // Waste teme for half a second - not nice code
0006E:  MOVLW  02
00070:  MOVWF  05
00072:  MOVLW  FA
00074:  MOVWF  06
00076:  MOVLB  0
00078:  RCALL  0004
0007A:  DECFSZ 05,F
0007C:  BRA    0080
0007E:  BRA    0084
00080:  MOVLB  F
00082:  BRA    0072
....................         output_low(PIN_B0);     // Switch pin B0 off - not nice code
00084:  BCF    F89.0
00086:  BCF    F84.0
```

```
13:20   Insert   Pjt: Test File   C:\Users\phil\Documents\PICC\UG Labs 18F\Test File.lst
```

The MPLAB IPE Environment

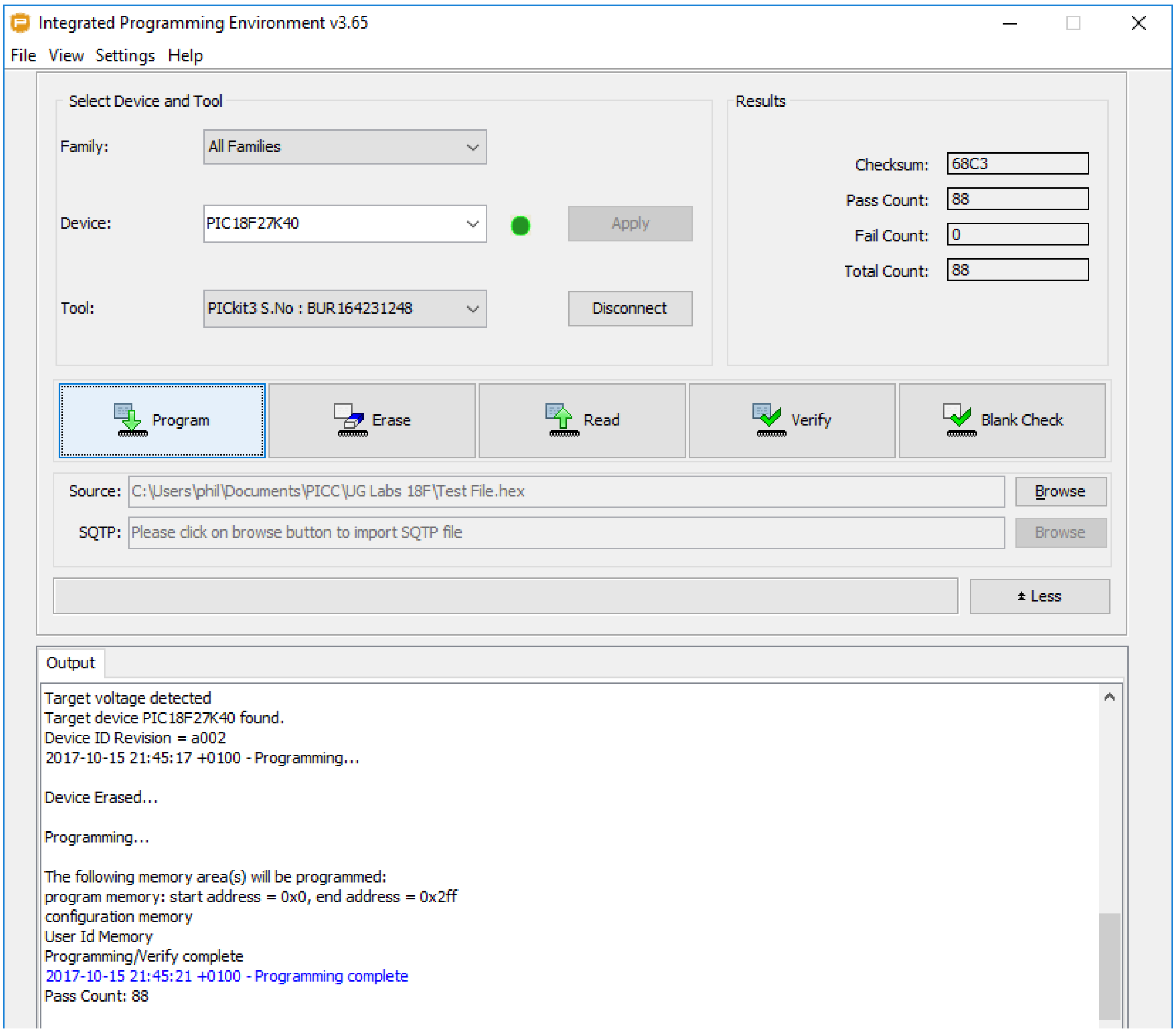
Type ‘MPLAB IPE’ in the Windows program search box.   Run the latest version of the Integrated Programming Environment installed on your PC (this is freely downloadable from Microchip along with some restricted C compilers).   First select the ‘Device’ to be a PIC18F27K40.   Now click the ‘File’ table, followed by ‘Import’ and then ‘Hex’.   Navigated to your Y drive where your source file is stored.   Always display the ‘details’ of when the file was created – the most common error is to accidentally have two copies of the same filename on your machine.   One file gets updated, whilst the other (unchanged file) gets repeatedly loaded to the device.   You should obtain a message in the output panel that the hex file has been loaded successfully.

Click the ‘connect’ button.   If the programmer detects a valid device and that it is powered, you will obtain a series of messages such as:

Target voltage detected
Target device PIC18F27K40 found.
Device ID Revision = a002

Click the ‘program’ button.   If everything is OK, you will obtain a screen such as that shown below.

Ignore any pop-up messages that warn you about destroying 3V3 devices if you accidentally program them with a 5 V supply.



Family: All Families

Device: PIC18F27K40

Tool: PICkit3 S.No : BUR164231248

Checksum: 68C3

Pass Count: 88

Fail Count: 0

Total Count: 88

Source: C:\Users\phil\Documents\PICC\UG Labs 18F\Test File.hex

SQTP: Please click on browse button to import SQTP file

```
Target voltage detected
Target device PIC18F27K40 found.
Device ID Revision = a002
2017-10-15 21:45:17 +0100 - Programming...

Device Erased...

Programming...

The following memory area(s) will be programmed:
program memory: start address = 0x0, end address = 0x2ff
configuration memory
User Id Memory
Programming/Verify complete
2017-10-15 21:45:21 +0100 - Programming complete
Pass Count: 88
```

Technical Information Notes

TN1. Internal Clock and Fuses

The oscillator module of the PIC 18F27K40 is show in Figure TN1.1. At first sight this appears bewilderingly complex. Matters get worse when one discovers that some blocks are switched in when the device is programmed using fuses, whilst others are programmed under software control at run-time.

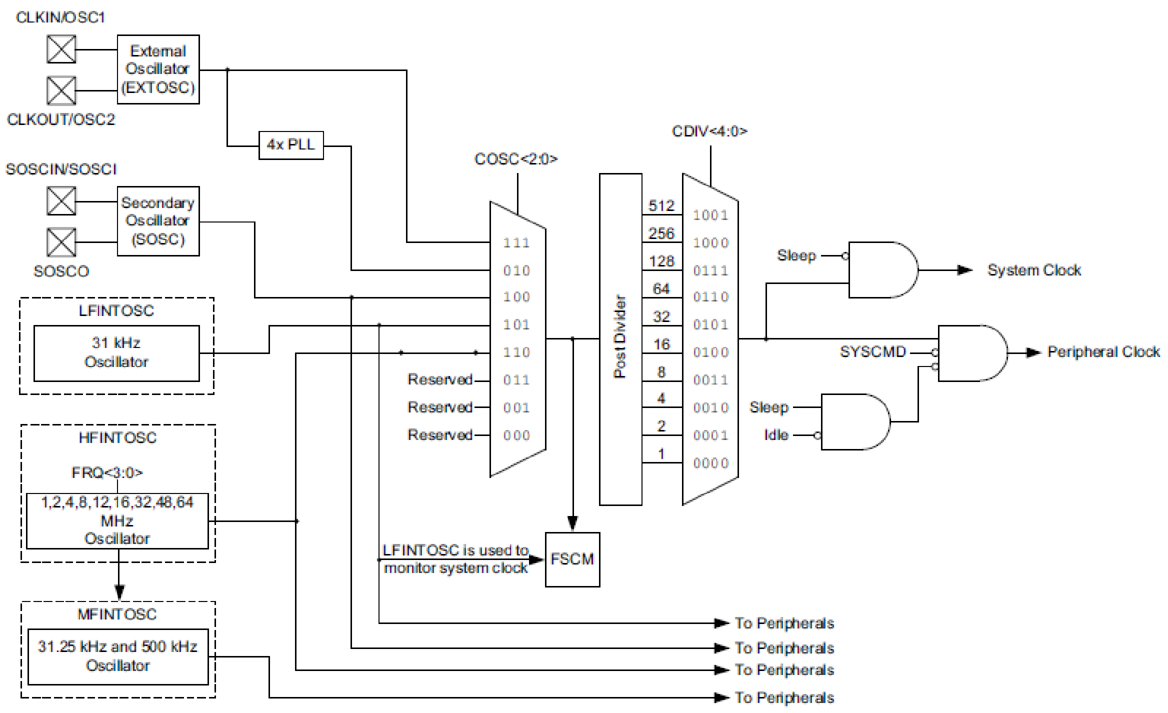
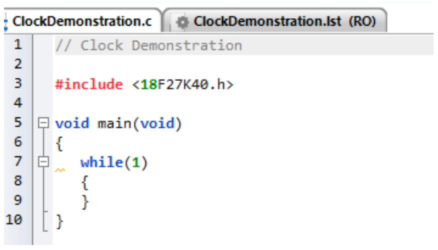


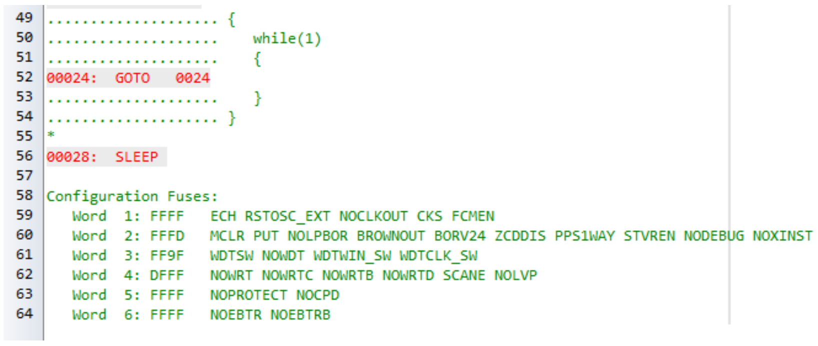
Figure TN1.1: PIC 18F27K40 Oscillator Module

It is probable that you may be responsible for routing these signals. The first stage is to determine what the compiler sets up for you by default. You might try a very simple program such as:



```
// Clock Demonstration

#include <18F27K40.h>

void main(void)
{
    while(1)
    {
    }
}
```

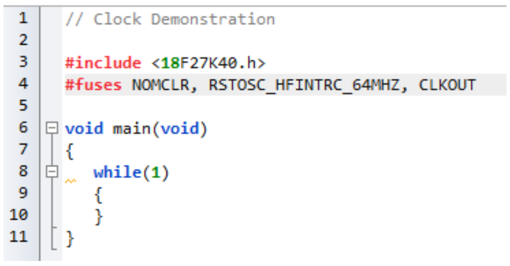
Then examine the compiled code to look at the default fuse settings. Open the '.lst' file and scroll down to the bottom of the file to see the fuses selected on your behalf.



```
.................... {
....................    while(1)
....................    {
00024:  GOTO   0024
....................    }
.................... }
*
00028:  SLEEP

Configuration Fuses:
   Word  1: FFFF   ECH RSTOSC_EXT NOCLKOUT CKS FCMEN
   Word  2: FFFD   MCLR PUT NOLPBOR BROWNOUT BORV24 ZCDDIS PPS1WAY STVREN NODEBUG NOXINST
   Word  3: FF9F   WDTSW NOWDT WDTWIN_SW WDTCLK_SW
   Word  4: DFFF   NOWRT NOWRTC NOWRTB NOWRTD SCANE NOLVP
   Word  5: FFFF   NOPROTECT NOCPD
   Word  6: FFFF   NOEBTR NOEBTRB
```

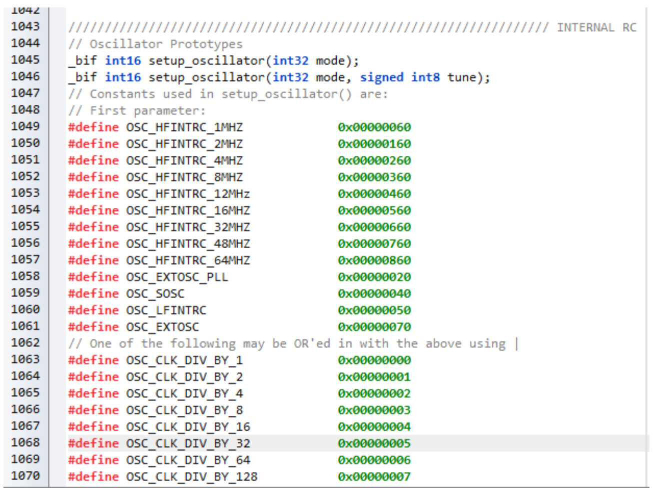
The Configuration Words are listed in Section 3.0 of the 18F27K40 data sheet. Now, examining this list of fuses in the 'lst' file shows that an external high speed clock option (ECH) has been selected on your behalf – not what you want. Similarly, the external reset pin is enabled (MCLR), whilst using the internal reset circuitry might be a good idea.

Right click on the 18F27K40 include file and select 'Open File at Cursor'. Now scroll down to the list of available fuses at about line numbers 18 to 32. Guess at suitable fuses and check their functionality in Section 3.0 of the data sheet. For example 'NOMCLR' to use the internal reset circuitry and 'RSTOSC_HFINTRC_64MHZ' to enable the 64 MHz internal oscillator. Here we find that the diagram shown in Figure TN1.1 is ambiguous as the implication is that the internal oscillator can run at a wide range of frequencies, whilst the fuse selections imply that only 4 MHz and 64 MHz is available. Finally, enable the clock output signal with the 'CLKOUT' fuse. So now, the basic program will look like:



```
// Clock Demonstration

#include <18F27K40.h>
#fuses NOMCLR, RSTOSC_HFINTRC_64MHZ, CLKOUT

void main(void)
{
   while(1)
   {
   }
}
```

The early parts of this experiment require a range of different clock frequencies to be used. Examining Figure TN1.1 under the assumption that the internal oscillator is running at a frequency of 64 MHz implies that some other form of oscillator selection must be available. Thus searching the include file reveals something interesting at line 1043 and Section 4.0 of the data sheet provides more information about the oscillator module. The implication is that the software command `setup_oscillator(int32 mode);` might be of use. A typical usage might be of the form `setup_oscillator(OSC_HFINTRC_64MHZ | OSC_CLK_DIV_BY_1);`. The second qualifier may be used to set the Post Divider ratio.

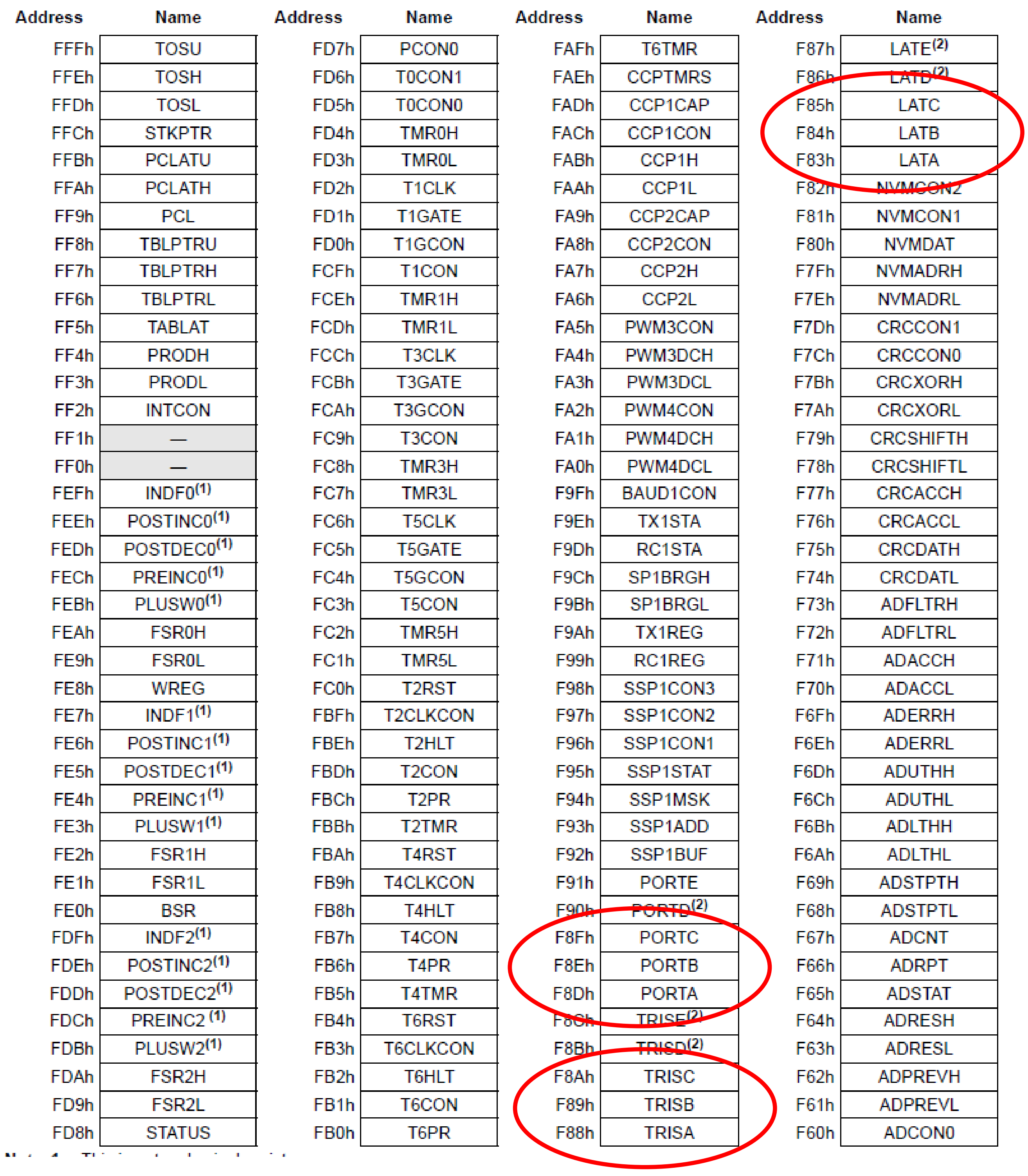


```
///////////////////////////////////////////////////////////////////// INTERNAL RC
// Oscillator Prototypes
_bif int16 setup_oscillator(int32 mode);
_bif int16 setup_oscillator(int32 mode, signed int8 tune);
// Constants used in setup_oscillator() are:
// First parameter:
#define OSC_HFINTRC_1MHZ            0x00000060
#define OSC_HFINTRC_2MHZ            0x00000160
#define OSC_HFINTRC_4MHZ            0x00000260
#define OSC_HFINTRC_8MHZ            0x00000360
#define OSC_HFINTRC_12MHz           0x00000460
#define OSC_HFINTRC_16MHZ           0x00000560
#define OSC_HFINTRC_32MHZ           0x00000660
#define OSC_HFINTRC_48MHZ           0x00000760
#define OSC_HFINTRC_64MHZ           0x00000860
#define OSC_EXTOSC_PLL              0x00000020
#define OSC_SOSC                    0x00000040
#define OSC_LFINTRC                 0x00000050
#define OSC_EXTOSC                  0x00000070
// One of the following may be OR'ed in with the above using |
#define OSC_CLK_DIV_BY_1            0x00000000
#define OSC_CLK_DIV_BY_2            0x00000001
#define OSC_CLK_DIV_BY_4            0x00000002
#define OSC_CLK_DIV_BY_8            0x00000003
#define OSC_CLK_DIV_BY_16           0x00000004
#define OSC_CLK_DIV_BY_32           0x00000005
#define OSC_CLK_DIV_BY_64           0x00000006
#define OSC_CLK_DIV_BY_128          0x00000007
```

Now, if a software command requires knowledge of the clock speed, the compiler will usually be told this by the programmer at compile-time using a compiler directive. For example, the command `delay_ms(1000);` will need to know the clock speed. This is achieved by using the `#use delay(clock=64MHZ,clock_out)` compiler directive. This directive will be active until another similar directive overrides it. **One of the biggest mistakes is to assume that compiler directives act like programming statements.** They are interpreted by the compiler irrespective of control statements such as `while`, `case`, `if`, `else` etc.

TN2. Memory-Mapped Structures

The EE2A lectures will have revealed that the IO ports are memory-mapped into the data memory space. First, the programmer must decide which pins of ports A, B and C are inputs and which are outputs – this information is derived by drawing the circuit diagram and examining the direction of each of the connections to the ports. The programmer then write a ‘1’ to the corresponding bits of the TRISA, TRISB and TRISC registers at locations 0xF88, 0xF89 and 0xF8A in order to make those pins inputs. Output pins are selected by writing a ‘0’ to the corresponding bits.



Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FD7h	PCON0	FAFh	T6TMR	F87h	LATE(2)
FFEh	TOSH	FD6h	T0CON1	FAEh	CCPTMRS	F86h	LATD(2)
FFDh	TOSL	FD5h	T0CON0	FADh	CCP1CAP	F85h	LATC
FFCh	STKPTR	FD4h	TMR0H	FACh	CCP1CON	F84h	LATB
FFBh	PCLATU	FD3h	TMR0L	FABh	CCP1H	F83h	LATA
FFAh	PCLATH	FD2h	T1CLK	FAAh	CCP1L	F82h	NVMCON2
FF9h	PCL	FD1h	T1GATE	FA9h	CCP2CAP	F81h	NVMCON1
FF8h	TBLPTRU	FD0h	T1GCON	FA8h	CCP2CON	F80h	NVMDAT
FF7h	TBLPTRH	FCFh	T1CON	FA7h	CCP2H	F7Fh	NVMADRH
FF6h	TBLPTRL	FCEh	TMR1H	FA6h	CCP2L	F7Eh	NVMADRL
FF5h	TABLAT	FCDh	TMR1L	FA5h	PWM3CON	F7Dh	CRCCON1
FF4h	PRODH	FCCh	T3CLK	FA4h	PWM3DCH	F7Ch	CRCCON0
FF3h	PRODL	FCBh	T3GATE	FA3h	PWM3DCL	F7Bh	CRCXORH
FF2h	INTCON	FCAh	T3GCON	FA2h	PWM4CON	F7Ah	CRCXORL
FF1h	—	FC9h	T3CON	FA1h	PWM4DCH	F79h	CRCSHIFTH
FF0h	—	FC8h	TMR3H	FA0h	PWM4DCL	F78h	CRCSHIFTL
FEFh	INDF0(1)	FC7h	TMR3L	F9Fh	BAUD1CON	F77h	CRCACCH
FEEh	POSTINC0(1)	FC6h	T5CLK	F9Eh	TX1STA	F76h	CRCACCL
FEDh	POSTDEC0(1)	FC5h	T5GATE	F9Dh	RC1STA	F75h	CRCDATH
FECh	PREINC0(1)	FC4h	T5GCON	F9Ch	SP1BRGH	F74h	CRCDATL
FEBh	PLUSW0(1)	FC3h	T5CON	F9Bh	SP1BRGL	F73h	ADFLTRH
FEAh	FSR0H	FC2h	TMR5H	F9Ah	TX1REG	F72h	ADFLTRL
FE9h	FSR0L	FC1h	TMR5L	F99h	RC1REG	F71h	ADACCH
FE8h	WREG	FC0h	T2RST	F98h	SSP1CON3	F70h	ADACCL
FE7h	INDF1(1)	FBFh	T2CLKCON	F97h	SSP1CON2	F6Fh	ADERRH
FE6h	POSTINC1(1)	FBEh	T2HLT	F96h	SSP1CON1	F6Eh	ADERRL
FE5h	POSTDEC1(1)	FBDh	T2CON	F95h	SSP1STAT	F6Dh	ADUTHH
FE4h	PREINC1(1)	FBCh	T2PR	F94h	SSP1MSK	F6Ch	ADUTHL
FE3h	PLUSW1(1)	FBBh	T2TMR	F93h	SSP1ADD	F6Bh	ADLTHH
FE2h	FSR1H	FBAh	T4RST	F92h	SSP1BUF	F6Ah	ADLTHL
FE1h	FSR1L	FB9h	T4CLKCON	F91h	PORTE	F69h	ADSTPTH
FE0h	BSR	FB8h	T4HLT	F90h	PORTD(2)	F68h	ADSTPTL
FDFh	INDF2(1)	FB7h	T4CON	F8Fh	PORTC	F67h	ADCNT
FDEh	POSTINC2(1)	FB6h	T4PR	F8Eh	PORTB	F66h	ADRPT
FDDh	POSTDEC2(1)	FB5h	T4TMR	F8Dh	PORTA	F65h	ADSTAT
FDCh	PREINC2 (1)	FB4h	T6RST	F8Ch	TRISE(2)	F64h	ADRESH
FDBh	PLUSW2(1)	FB3h	T6CLKCON	F8Bh	TRISD(2)	F63h	ADRESL
FDAh	FSR2H	FB2h	T6HLT	F8Ah	TRISC	F62h	ADPREVH
FD9h	FSR2L	FB1h	T6CON	F89h	TRISB	F61h	ADPREVL
FD8h	STATUS	FB0h	T6PR	F88h	TRISA	F60h	ADCON0

[illegible]

Figure TN2.1: Special Function register Map

The programmer may read the values of input pins by reading the contents of the corresponding bits in the PORTA, PORTB and PORTC registers located at 0xF8D, 0xF8E and 0xF8F.

The programmer may write the values of output pins by writing the contents of the corresponding bits in the LATA, LATB and LATC registers located at 0xF83, 0xF84 and 0xF85.

It will be noticed that the three ports are always mapped to three adjacent bytes in the memory map. It will also be noticed that sometimes the experiment calls for a single-bit, sometimes for six-bits and sometimes for two groups of three-bits talking to the same location (e.g. activating LEDs). Now a 'structure' can act as a convenient container to wrap data items of different bit-widths into adjacent memory cells. For example, define a structure using any definition name you choose, in this case `IO_def1`:

```
struct IO_def1
{
   int  ExperimentSelection:4;   // Pins RA0..3: Four switches select experiment
   int  unused_A1:2;             // Unused pins RA4..5
   int1 Clock_Output;            // RA6 clock output
   int1 unused_A2;               // Unused pin RA7
   int  Logic_Input_A:2;         // Pins RB0..1 pins connected for bitwise logic
   int  Logic_Input_B:2;         // Pins RB2..3 pins connected for bitwise logic
   int  unused_B:4;              // Unused pins RB4..5, RB6..7 to programmer
   int1 Single_Bit_LED;          // Pin RC0 LED used for single-bit indication
   int  unused_C:7;              // Remaining bits of Port C
};
```

Attempt to read the definition structure as a list of signals described in ascending order. Single bit variables are describes as `int1`, whereas multibit variables use the colon to define the word length, e.g. a two-bit variable might be described using `int  Logic_Input_A:2;`.

The next stage is to declare three variables corresponding to PORT, LAT and TRIS. Normally, the compiler would allocate space for these three variables in the first available cell in the data memory.

```
struct IO_def1 IO_Port_1;
struct IO_def1 IO_Port_1_Latch;
struct IO_def1 IO_Port_1_Direction;
```

In this case, we wish to over-ride the compiler and force it to allocate the variables to the desired cells in the data memory space corresponding to the IO ports. This is achieve using another compiler directive (`#byte`).

```
#byte IO_Port_1 = 0xF8D            // PORTA register in SFR
#byte IO_Port_1_Latch = 0xF83      // LATA register in SFR
#byte IO_Port_1_Direction = 0xF88  // TRISA register
```

For different sections of the experiment, we need different word-length variables talking to the same ports. Thus we use multiple structure definitions, declaration and memory-mappings. An example of the third structure definition is as follows:

```
struct IO_def3
{
   int  ExperimentSelection:4;   // Pins RA0..3: Four switches select experiment
   int  unused_A1:2;             // Unused pins RA4..5
   int1 Clock_Output;            // RA6 clock output
   int1 unused_A2;               // Unused pin RA7
   int1 Set;                     // Pin RB0 connected to switch for 'Set'
   int1 Reset;                   // Pin RB1 connected to switch for 'Reset'
   int  unused_B:6;              //
   int  Multi_Bit_LED_Group1:3;  // Pins RC0..3 - LEDs used for binary counters
   int  Multi_Bit_LED_Group2:3;  // Pins RC4..6 - LEDs used for Gray counters
   int  unused_C:2;              // Remaining bits of Port C
};
```

Following any variable definitions within the '`main`' function, the programmer will define the direction of each IO pin.   Typical code might look like:

```
   IO_Port_1_Direction.ExperimentSelection = 0b1111;  // Pins B4..7 as input
   IO_Port_1_Direction. unused_A1 = 0b00;             // Unused pins as output
   IO_Port_1_Direction.Clock_Output = 0b0;            // CLKOUT is an output
   IO_Port_1_Direction.unused_A2 = 0b0;               // Unused pin as output
   IO_Port_1_Direction.Logic_Input_A = 0b11;          // RB0..1 as inputs
   IO_Port_1_Direction.Logic_Input_B = 0b11;          // RB2..31 as inputs
   IO_Port_1_Direction.unused_B = 0b0000;             // Unused pins as output
   IO_Port_1_Direction.Single_Bit_LED = 0b0;          // Single LED as output
   IO_Port_1_Direction.unused_C = 0b0000000;          // Unused pins as output
```

Later in the executable sections of the code the programmer might wish to act on an input variable using code such as:

```
      // Select which experiment to run
      switch (IO_Port_1.ExperimentSelection)
```

Or output values to pins using code such as:

```
      IO_Port_1_Latch.Single_Bit_LED = 0b1;        // Single LED on
      delay_ms(500);
      IO_Port_1_Latch.Single_Bit_LED = 0b0;        // Single LED off
      delay_ms(500);
```

TN3. C Syntax

This first experiment is designed as a refresher for the C language and as an introduction to a variety of embedded controller enhancements. A suitably concise guide to the C language syntax may be found at https://en.wikipedia.org/wiki/C_syntax.

You will need to use arrays for both input and output mapping. Remember that arrays may be initialised by using a syntax such as:

```
int LUT[64] = {0x00,0x01,0x03,0x02,0x06, ………
```

The contents of this look-up table may be accessed using a syntax such as:

```
IO_Port_2_Latch.Multi_Bit_LED = LUT[Binary_Counter];  // Output to Port C LEDs
```

The contents of this look-up table are used to convert from a pure binary code to a Gray code. You could spend a happy hour typing in the conversion values, or you could improve your programming skills by automatically filling the array under program control. I used Matlab to generate a text file that was then cut-and-pasted into the main C program. The Matlab syntax for a similar table output would be:

```
fprintf('const int LUT[64]={')
fprintf('0x%s',dec2hex(round(WaveTable(TableIndex)),2)) % 2-digit hex number
fprintf('};')
```

Checklist to Obtain a ‘W’ during Demonstration

Switch Code	**Comment**	**Seen**
0000	Demonstrate that clock microprocessor is running at 4 MHz, pin RA6 produces 1 MHz	
0001	Demonstrate that clock microprocessor is running at 8 MHz, pin RA6 produces 2 MHz	
0010	Demonstrate that clock microprocessor is running at 16 MHz, pin RA6 produces 4 MHz	
0011	Demonstrate T-type flip flop in toggle mode. Pin RC0 produces 1 Hz	
0100	Demonstrate that Port C0..5 implements a 6-bit pure-binary up-counter.	
0101	Demonstrate that Port C0..5 implements a 6-bit pure-binary down-counter.	
0110	Demonstrate an output look-up table via a 6-bit Gray-code up-counter	
0111	Demonstrate multiple output variables: 3-bit pure-binary up-counter and 3-bit Gray-code up-counter.	
1000	Demonstrate bitwise AND operation.	
1001	Demonstrate bitwise OR operation.	
1010	Demonstrate bitwise NOT operation.	
1011	Demonstrate bitwise Exclusive-OR operation.	
1100	Demonstrate set-reset flip-flop synthesis.	
1101	Demonstrate input look-up table.	
1110	Demonstration of ‘bright-eyed and bushy tailed capability’	
	All the above controlled by four switches and a ‘switch/case’ construct.	
	Note: The above tasks will require multiple overlaid structures mapped to the same IO pins.	

Name:

EE2A Experiment 2

PIC Introduction

Feedback Mark Sheet – Paste into the logbook

The following will be assessed during the laboratory session (PGTA to circle, date and sign)

Circuit construction and demonstration:	**Yes**	**Partially**	**No**
Demonstration of sixteen programming constructs linking finite state machines and embedded controllers (35%)	**W**	**WP**	**NW**

Inspection Mark For Log Book:	**Could not be better**	**Good attempt**	**Room for improvement**	**Appalling**
A 'flick-test' of the log book will be carried out. This will be based on the purpose of the log book – to convey useful information to other engineers and to allow others to carry on with the work.	**4**	**3**	**2**	**1**

At the end of the year, the log-book will be handed in and assessed using the following criteria:

Preparatory Work:
Evidence of adequate preparatory work undertaken outside laboratory (15%). Students to have decided what is 'adequate' as part of their work.
Source Code:
Existence and completeness of the program header (author, date, filename, target device, fuse settings, program function) (5%). Appropriateness and clarity of comments, labels and variable/constant definitions (5%). Efficient use of code (i.e. no redundancy) (5%). Program Elegance (5%). Technical content (10%). Code print-outs are assumed.
Reflective journal:
Care and neatness of preparatory work (written up outside lab) (5%). Sensible attempt at keeping a log-book (written up at the time) to convey engineering information to other professionals. (10%). If this student was a professional engineer who left his/her organization today, could another engineer pick up the pieces in six months time ? Conclusions- sensible executive summary & record of the learning experiences gained by the student during this experiment. (5%).

Verbal feedback will be provided during the laboratory.

Timeliness (Autumn Term)

Week in which experiment was demonstrated to supervisor (shaded blocks show overrun)

Week 4	Week 5	Week 6	Week 8	Week 9	Week 10	Week 11
	Excellent Progress	Excellent Progress	Excellent Progress	Good Progress	Average Progress	Getting Worried