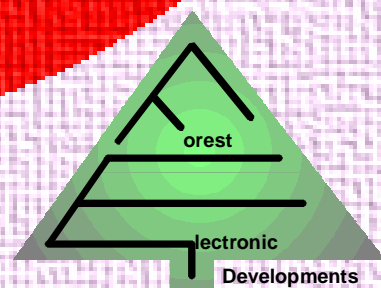


Learn to Use C

***with the Forest Electronic
Developments
PIC C Compiler***



THE FED C Compiler is intended for all serious programmers of the PIC who would like the convenience of a high level language together with the speed of assembler. With our C Compiler you no longer have to worry about ROM and RAM paging, you can call to a depth limited only by RAM not by the 8/32 level call stack, use 16 and 32 bit arithmetic types for full precision, and use any of our standard library routines for general purpose data handling and interfacing.

A free version is available to compile all the examples in this manual, you can download this from :

http://www.fored.co.uk/html/wiz-c_download.html

This manual is copyright (C) Forest Electronic Developments 2000, 2008. It may not be copied, transmitted to 3rd parties in any form, or altered without express permission of Forest Electronic Developments.

Forest Electronic Developments can accept no responsibility for the consequences of any errors or omissions in this introductory manual

Forest Electronic Developments

12 Buldowne Walk

Sway

Lymington

HAMPSHIRE

SO41 6DU

Sales : +44 - (0)1590 - 681512

info@fored.co.uk

Or see the **Forest Electronic Developments** home page
on the world wide web at the following URL:

<http://www.fored.co.uk>

Contents

LEARN TO USE C WITH THE FOREST ELECTRONIC DEVELOPMENTS PIC C COMPILER	1
CONTENTS	1
CONTENTS	2
CONTENTS	3
1 INTRODUCTION	5
2 THE FED DEVELOPMENT BOARDS	7
3 PIC PORTS.....	13
4 DOWNLOADING THE WIZ-C PROFESSIONAL COMPILER – WIZ-C PROFESSIONAL.....	15
5 OUR FIRST PROGRAMS WITH WIZ-C PROFESSIONAL	16
6 COMMENTS AND STATEMENTS	29
7 VARIABLES AND CONSTANTS	30
8 EXPRESSIONS	41
9 FUNCTIONS	49
10 PROGRAM CONTROL.....	57
11 POINTERS.....	64
12 STRUCTURES & UNIONS	70
13 THE PRE-PROCESSOR	74
14 SUPPORT FOR THE PIC	77
15 REAL TIME PROGRAMMING EXAMPLE	78
16 WIZ C PROFESSIONAL – APPLICATION DESIGNER.....	94
APPENDIX A - REAL TIME PROGRAMMING SOURCE	111
APPENDIX B - LIST OF FED PIC KEYWORDS	118
APPENDIX C - LIST OF WIZ-C PROFESSIONAL STANDARD LIBRARY FUNCTIONS.....	119
APPENDIX D - LIST OF SUPPORTED PIC'S AT DECEMBER 2007	121

APPENDIX E – CONFIGURATION FUSES 122

1 Introduction

Welcome to WIZ-C PROFESSIONAL, this manual will take you through the process of learning C from variables through constants to pointers and then structures and unions. Most of the examples are standalone and are as small as possible to enable the purpose and effect to be easily understood. Nearly all can be run on the WIZ-C Professionalsimulator so you can experiment quickly. We recommend you open this document in a word processor and cut and paste the examples into PIC C as you come to those examples you'd like to run or modify yourself.

FED offer a free C compiler which can compile programs up to 2K words in size (the full version can handle the full range). This compiler is called WIZ-C professional.

You are recommended to work through the first example as it will show you how to use the simulator and waveform analyser which will allow you to check out all the example programs for real.

Most examples in the earlier part of the manual are actually not PIC dependant - but towards the end we will look at a real program using interrupts, clocks, an LCD display and buttons for input, and which uses the ports on the PIC.

We assume an elementary knowledge of programming in high level languages - in particular the notion of variables and arrays and we assume a knowledge of bits and bytes, ROM and RAM and hex and binary representations.

A knowledge of the PIC architecture is not necessary, however only simple Port operations and use of Timer 0 are covered in this manual. We strongly recommend that readers also look at the Microchip PIC data sheets (which are included on the WIZ-C Professional CD-ROM).

Function pointers are not covered although they are supported by WIZ-C PROFESSIONAL. The WIZ-C Professional manual explains differences between WIZ-C Professional and ANSI C - there are actually few divergences from the standard.

Readers are recommended to read "The C programming language" by Kernigan and Ritchie to supplement this introductory text.

All of the examples in this manual are available on the web from FED at :

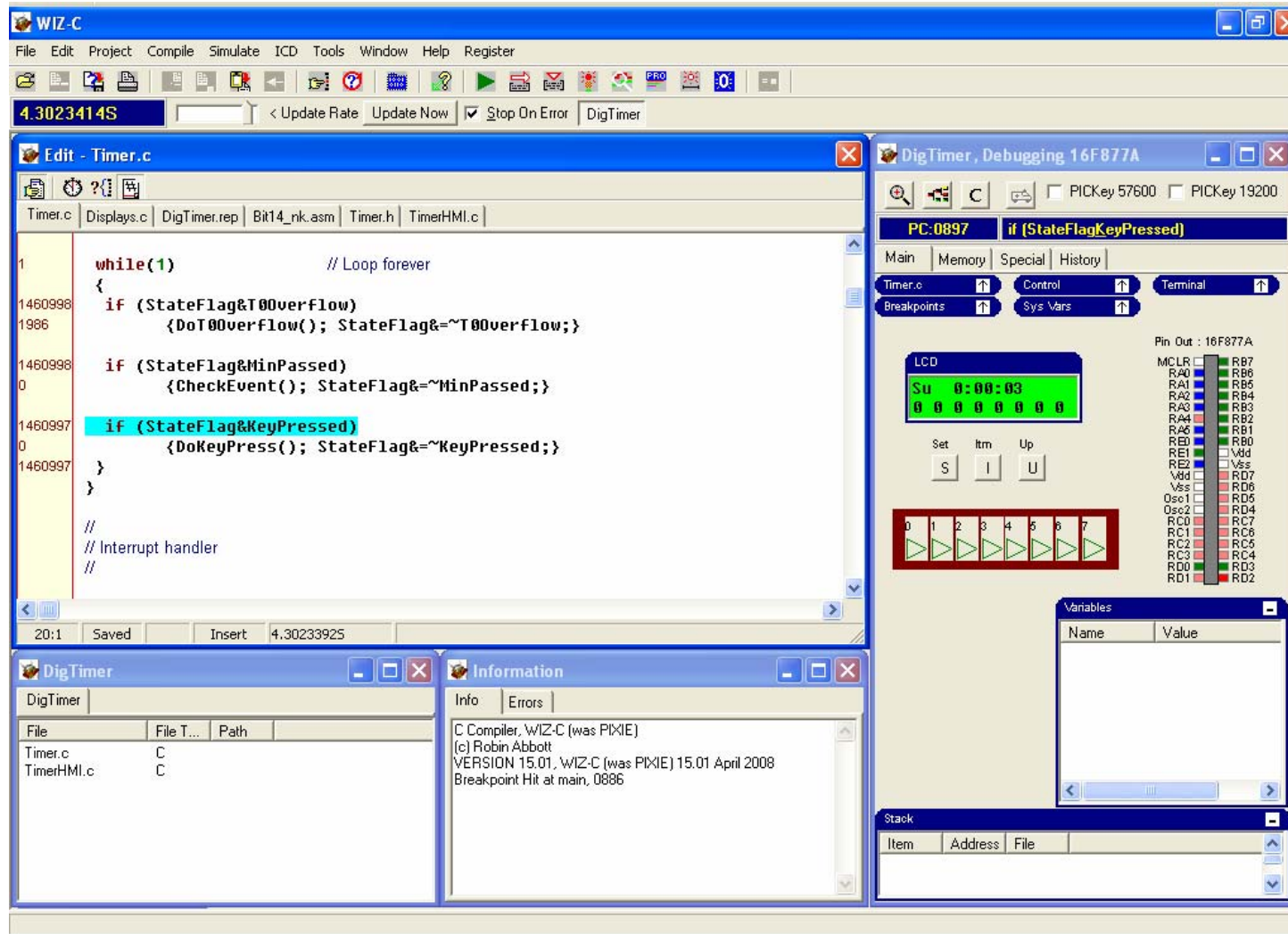
www.fored.co.uk

Look for the support and downloads page from the top level page.

Menus – Please note throughout this manual a menu option is shown in the following format :

File | Open

This means click the File menu then click the Open item on that menu.



WIZ-C Professional Compiler – running the Chapter 14 example

2 The FED Development boards

FED have a number of development boards on which all of our examples can be run. They can also be built up on vero-board or similar prototyping systems.

These are presented here for information,

2.1 *Duetronix 18 development board*

FED have a simple development/evaluation board for 18 pin devices which is suitable for use with all the examples and the tutorial. The board has the following features:

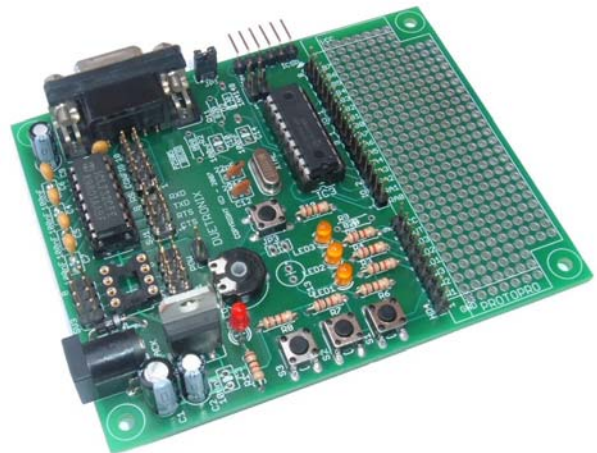
- 780X Based on Board PSU
- Power on LED
- Reverse polarity protection
- 3 Push Buttons,
- 3 LEDs
- MAX 232 based RS232 Serial Communications + DB 9 Connector
- ICSP Header and supporting circuitry
- Prototyping area with all I/O tracked to parallel connection strip
- PCB Dimensions (10 cm x 8 cm)

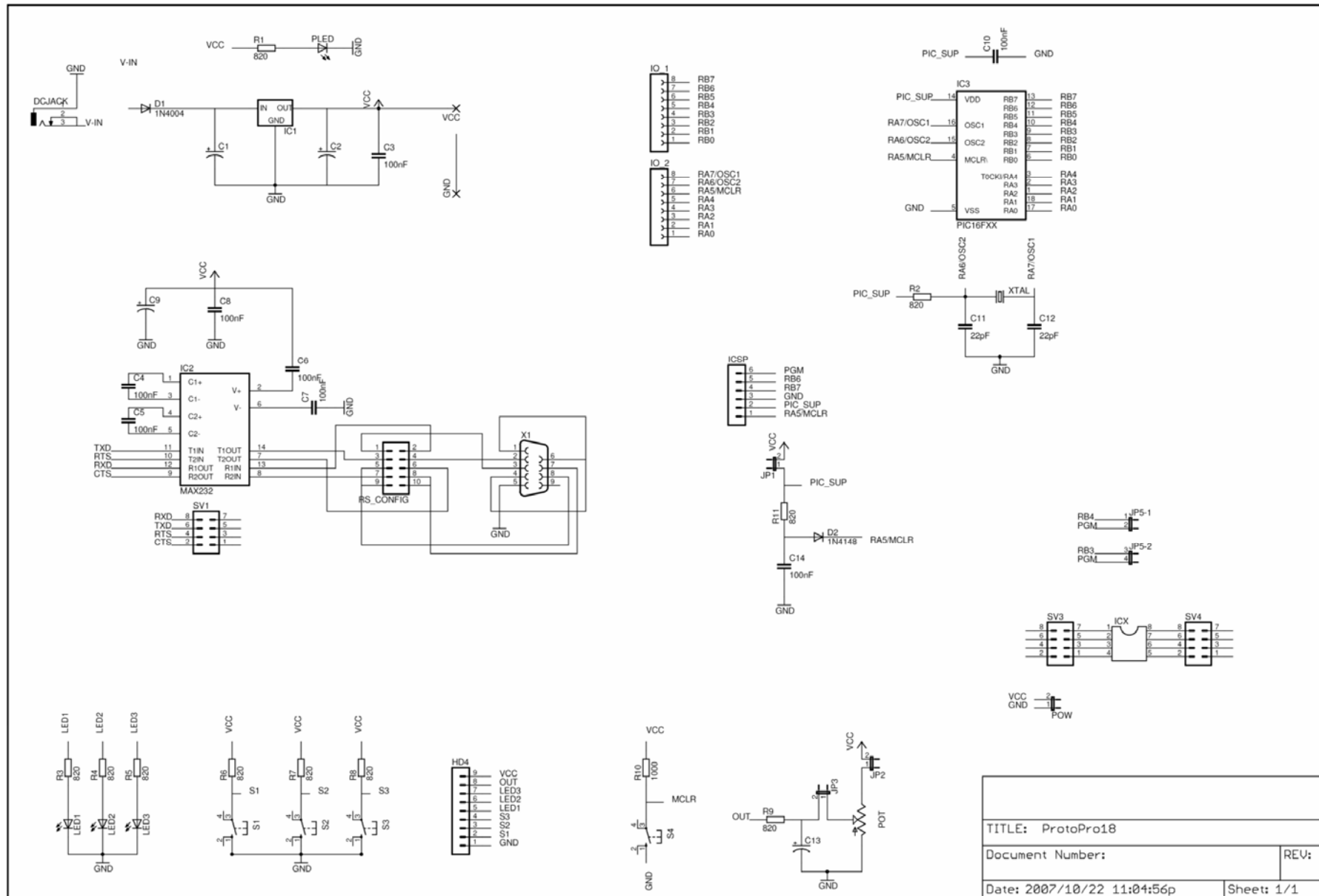
The circuit diagram is shown below:

The board details and further information

are available here :

http://www.fored.co.uk/html/duetronix_18.html





2.2 40 pin PIC development board

FED have a capable development/evaluation board for 40 pin devices such as the 16F877 or the 18F452 which is suitable for use with all the examples and the tutorial and full example program. The board has the following features:

- Supports all 40 pin 16 series PIC's.
- Will run FED PIC BASIC interpreter/development system (supplied with board)
- Has on board 5V, 1A, regulator
- 2 Serial interfaces for standard 3 wire serial communications connected to PIC serial interface and CCP pins
- Supports 8 pin EEPROM/RAM devices with IIC interface
- 20MHz crystal oscillator
- 32 I/O pins (all except E2) available on standard IDC connectors
- Connector for LCD module - 1:1 pin out including brightness control
- Key pad connector for 4x4 hex keypad
- Full In Circuit Serial programmer - connect to the PC and program hex files direct to any of the 40 pin PIC's
- 4 LED's
- LCD, Hex keypad and LED's all connect to PORTD/PORTE leaving all other ports free for other I/O functions

The circuit diagram is shown below:

Further details and purchase information are available from here :

<http://www.fored.co.uk/html/devboard.HTM>



2.2.1 Connections for the FED development board:

J1 - Serial Connector 1, Programming+CCP

2	Tx Data (from board)
3	Rx Data (to board)
5	Ground

J2 - Serial Connector 2,
PIC Asynchronous serial port

2	Tx Data (from board)
3	Rx Data (to board)
5	Ground

Conn 1 - Keypad connector

1	Keypad column 0
2	Keypad column 1
3	Keypad column 2
4	Keypad column 3
5	Keypad row 0
6	Keypad row 1
7	Keypad row 2
8	Keypad row 3

Conn 2 - LCD Module

1	Ground	LCD Module pin 1
2	+5V	LCD Module pin 2
3	Contrast	LCD Module pin 3
4	RS	LCD Module pin 4
5	R/W	LCD Module pin 5
6	E	LCD Module pin 6
7	NC	
8	NC	
9	NC	
10	NC	
11	D4	LCD Module pin 11
12	D5	LCD Module pin 12
13	D6	LCD Module pin 13
14	D7	LCD Module pin 14

P1 - Power

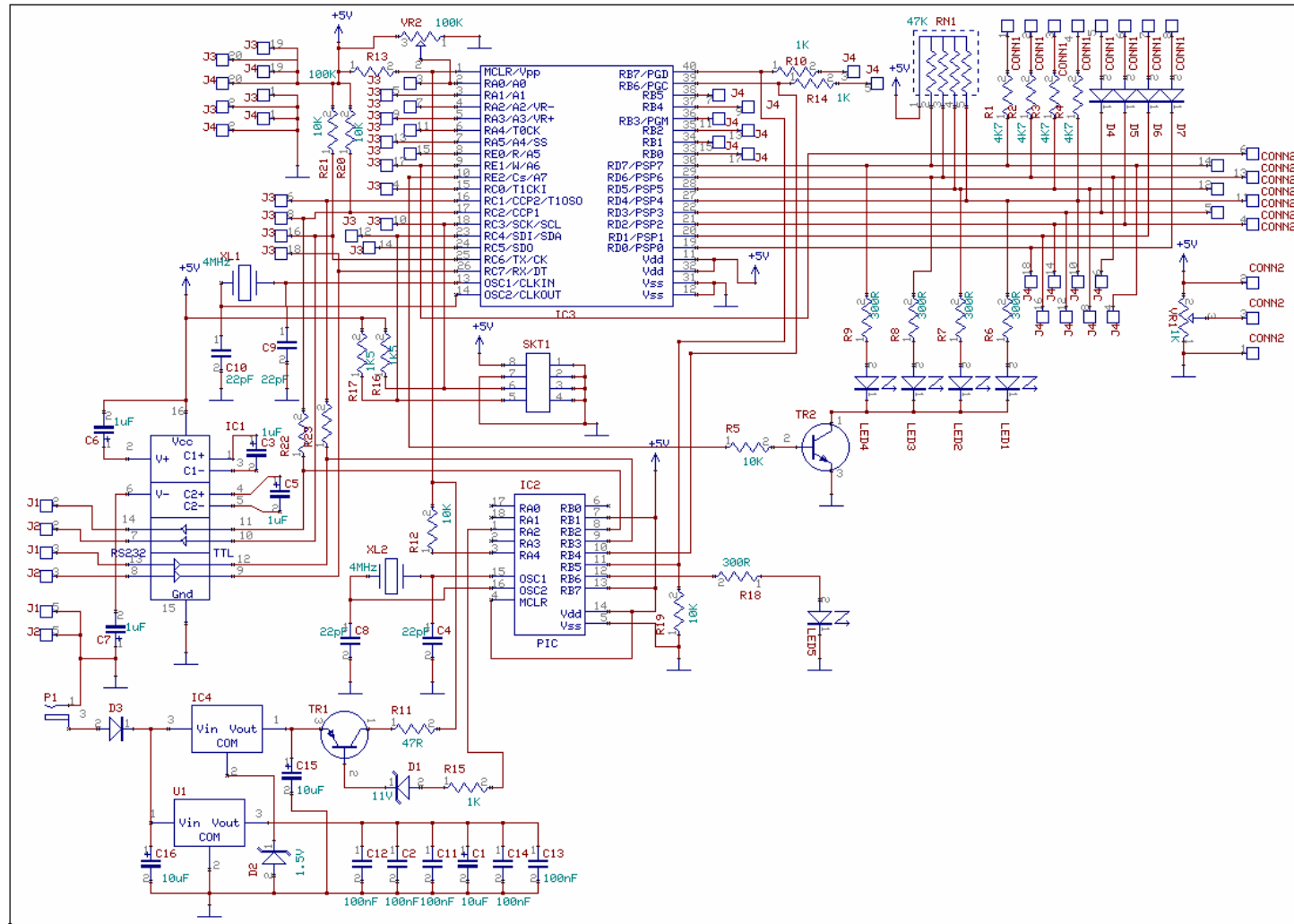
1	Ground
2	NC
3	+9-30V

J3 - PIC Ports A, C & E

1	Ground
2	Ground
3	RA0
4	RC0
5	RA1
6	RC1
7	RA2
8	RC2
9	RA3
10	RC3
11	RA4
12	RC4
13	RA5
14	RC5
15	RE0
16	RC6
17	RE1
18	RC7
19	+5V
20	+5V

J4 - PIC Ports B & D

1	Ground
2	Ground
3	RB7
4	RD7
5	RB6
6	RD6
7	RB5
8	RD5
9	RB4
10	RD4
11	RB3
12	RD3
13	RB2
14	RD2
15	RB1
16	RD1
17	RB0
18	RD0
19	+5V
20	+5V



3 PIC Ports

If you already understand how the PIC external ports operate then you can skip this section and move on to the next, but it is worth reviewing here.

The PIC's have a number of pins which can be used as general purpose inputs, or outputs, or can be switched between modes during operation.

If you look at a PIC data sheet you will notice that the 18 pin devices such as the 16F84 have pins labelled as RA0, RA1, and RB0, RB1 etc. Sometimes these pins may be shared with other pin functions, but nearly all external data pins belong to ports and are named in this way.

Pins labelled RA0, RA1 etc. all belong to Port A, The pin labelled RB0 belongs to Port B and so on until RJ0 on some of the larger PIC's which have up to 9 ports. RB0 is bit 0 of port B, so if we read Port B then we will read an 8 bit value of which bit 0 is the value of RB0, bit 1 will be the value of RB1 etc. Similarly when writing a value to Port B, then an 8 bit value is written, bit 7 will be written to RB7 for example.

So to set RB0, RB1, RB2 and RB3 high (+5V), whilst clearing (0V) the upper 4 bits of port B then port B would be set to the hex value 0F.

The ports are referred to in almost every language used to program the PIC (including WIZ-C PROFESSIONAL) by the names PORTA, PORTB, PORTC etc. Thus to set the value of PORTB to 0F (which is 15 in decimal), we would use (in C):

```
PORTB=15;
```

The semi-colon is used in C to show the end of an action.

As each port is bi-directional it has a control register which tells the PIC whether each pin is an input or an output. These control registers are called the tri-state registers, and there is one tri-state register for each port. The tri-state register for Port A is called TRISA, for Port B is called TRISB etc. When a bit is set to a 0 then the corresponding port pin is set to an output, when it is set to 1 then the bit is an input. For example if we wanted to make Port B bit 0 an input (this is pin RB0), but leave all the other pins of Port B as outputs then we would use the following code (in C) :

```
TRISB=1
```

The table below shows various examples of how PIC port pins may be set.

Action	Code to set up the Port Pins :
Set all pins of Port B to outputs	TRISB=0;
Set all pins of Port C to inputs	TRISC=255;
Set RB0, RB1 and RB4 to outputs, all other pins of Port B to inputs	TRISB=0xEC; Note that this is how to write a hex value in C - the value is preceded with "0x"
Set Port A to value 7	PORTA=7;
Read Port B to variable x	X=PORTB;

If an output pin is read then the value read will be the last value written to that output. If an input pin is written then this will have no effect on the PIC pin.

When the PIC is reset, all of the TRIS registers are set to 1's, so that on a reset each pin of the PIC is an input.

Note – The 18xxxx series of devices have LAT registers which may be used instead of PORT – the reason for doing this which is related to read/modify/write instructions is explained in the 18 series data sheet. For the moment the PORT variable may be used

3.1 Important note for the use of ports on devices with A/D conversion

Several devices including the 16F877 have internal Analogue to Digital converters which operate on port pins (for the 16F877 these are RA0,RA1,RA2,RA3,RA5,RE0,RE1,RE2). When the chip is reset all of these pins are configured to act as analogue inputs and cannot be used for digital inputs or outputs. In our program we may wish to make use of these pins for digital I/O. For most devices the following C line will set all of the A/D pins to be digital inputs or outputs, it should be included in any program you write where the A/D converter is not required.

```
ADCON1=7 ;
```

The data sheet should be consulted for other values for ADCON1 where it is possible to define some inputs as A/D and the other as digital.

4 Downloading the WIZ-C Professional Compiler – WIZ-C Professional

4.1 Introduction

The Forest Electronics ANSI C Compiler is called WIZ-C Professional. You may have known WIZ-C as PIXIE previously – the change in name is for copyright reasons.

It allows you to select software library components (*elements*) from a palette, set their parameters by drop down lists, check boxes and verified text entry. It will generate the main application, initialisation code and main loop automatically and considerably speeds the front end development of PIC projects.

WIZ-C includes a full C development and simulation environment with a Rapid Application Designer (RAD) front end. For the purposes of this manual we will configure WIZ-C not to use the application designer in which case it behaves exactly like the WIZ-C Professional Compiler. The last chapter of the book looks at the use of the application designer which considerably eases the development of complex projects.

4.2 Downloading and Installation of the free version

The free version is downloaded from the following link :

http://www.fored.co.uk/html/wiz-c_download.html

It is supplied as a zip file. Extract all the files to a temporary directory. Run Setup.exe to install the program. It is strongly suggested that (at least initially) the program is installed in the default directory which will allow the example projects to operate correctly. The demonstration free version will compile applications of up to 2000 Words, it can be upgraded to the unlimited full version from here :

http://www.fored.co.uk/acatalog/WIZ_C_PIC_Visual_Development_in_C.html

The free version can also be unlocked once payment has cleared to give the full functionality of the complete version, however FED will always send a CD with the full version.

4.3 Installation of the full version

The program is installed from CD-ROM. For the CD-ROM insert into the CD drive and an opening menu should come up. Alternatively run the program "SETUP.EXE" from the CD.

The manual is supplied as an Adobe Acrobat (PDF) Format file, a copy of Acrobat is supplied on CD-ROM and can be installed from the opening menu. The manual is duplicated in the help files which are accessible under the Help menu.

4.4 Support

Users of the full version can purchase contracted support solutions – details available from our web site :

<http://www.fored.co.uk/html/consupport.htm>

5 Our first programs with WIZ-C PROFESSIONAL

The very first program that we will write will simply flash an LED on and off on a port. We'll present two versions of the program, the first for the 16F84 and the second for the FED 16F877 development board.

Initially we'll run through the processes step by step and then for later chapters we'll simply present the programs – the process of starting a new project, compiling, and debugging will be identical for all the examples within this manual.

Menus – Please note throughout this manual a menu option is shown in the following format :

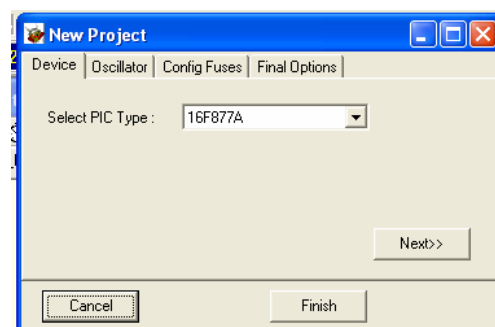
File | Open

This means click the File menu then click the Open item on that menu.

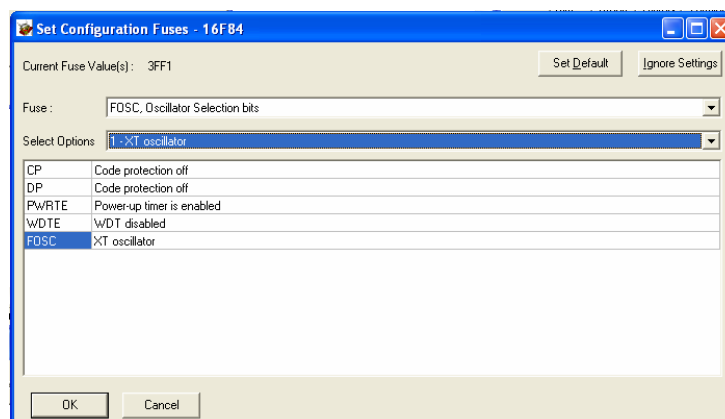
5.1 Start the program and open a new project

Start WIZ-C Professional by double clicking the PIC C icon within the PIC C submenu of the Start menu on your computer.

To open a new project then use the **Project | New Project or Project Group...** menu item. This brings up the new project wizard :

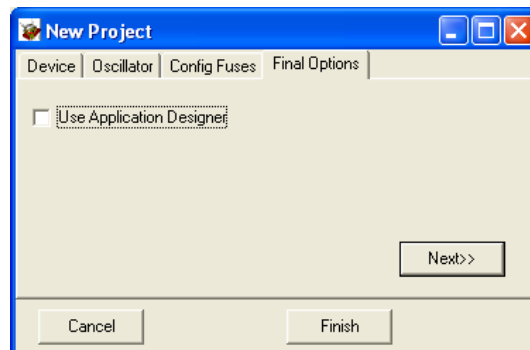


Use the “Select PIC Type” box to select the 16F84 device. Click the next button. This will bring up a number of standard oscillator frequencies – select 4MHz. Click Next, this brings up a button allowing the configuration fuses to be set. Configuration fuses control items such as the oscillator type and the use of the watch dog timer. It is normally essential to set the configuration fuses, click the button to see the options. This is how we set it :



The use of configuration fuses is covered in much greater detail in Appendix E – you might like to consult this now if the box above makes no sense.

Click OK and then Next to display the final options. By default we would like to turn the application designer off so **uncheck this option** :



Finally press Next for the last time (or Finish).

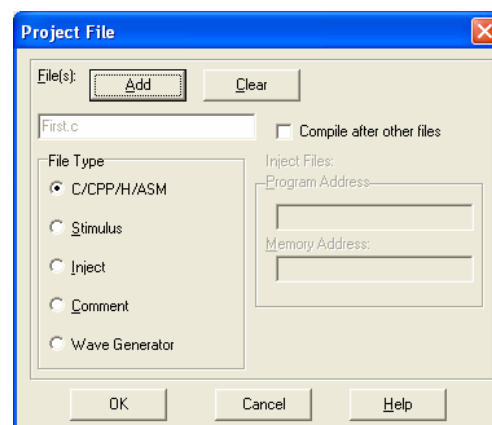
A File dialog box will be brought up to allow the project to be saved. Select a suitable directory – for example “C:\Program Files\FED\PIXIE\Projects\Tutor” will be installed as a blank directory when the main program is installed. If you want to create a new directory which does not exist create it using the New Folder button on the Open Dialog. Enter the file name "First", click Open. As this is a new project it may appear in minimal form, maximise the window by using the button in the top right of the window bar and then use the **Window | Arrange for Edit** option to position the new project on screen (you could also press ALT+E).

Call the new directory "First". Select the new directory, double click to enter it, and then enter the project file name which we will use for our new project - "First". Press OK.

FED recommend maximising the window, and then using the **Window | Arrange neatly** menu option to arrange the windows on screen.

Initially this project will consist of one C file. Use **File | New** to create a blank file, now use **File | Save As** to select a file name. This will bring up a file open dialog box. No files will be present so enter the filename “First” and click on OK.

Select the project window (the window which has the title “First”) this will be blank so click the window and press the Insert key. This will bring up a file open dialog box, navigate to your First directory and one file "First.C" will be present - select this file and click on OK. A dialog box will now appear with this file name and a number of options, we can define files as being C files, or as a comment file. In this case the file is used for compilation, so make sure that the file type C is selected and click OK :



Double click the file to open it, now enter the program, this first program is for the 16F84, the version for the 16F877 is shown later in the chapter. We can simulate this program so we don't need to run it on a development board. The program will flash an LED connected to Port B, Bit 0.

Program 1 – example for the 16F84:

```
//
// Example program - LED Flasher
//
#include <pic.h>           // Includes all relevant PIC definitions
#include <delays.h>

void main()
{
    TRISB=0xfe;
    while(1)
    {
        Wait(250);
        PORTB=PORTB^1;
    };
}
```

We'll look at each line of this program in detail later, for the moment enter it exactly as shown including semi-colons and then save the file using the **File | Save** menu option. You can also use the button on the tool bar :



5.2 Compile the project

The application may now be generated for the first time. To do this use the **Compile | Compile** Menu, or press Control and the F9 key, or use the small button on the tool bar



A box titled "Compiler Options" will be shown, this allows the C Compiler options to be set. This box will appear the first time that a project is compiled, but will not appear again. To set the options after the first time then use the **Project | Current Project Options** menu option.

As we don't need to alter the default options then this can be ignored so click OK and the project will compile and assemble. At present we still have some code to define. The information window should show the progress of the compilation, all being well the project will compile OK, this project takes a total of 85 program words – nearly all of this is initialisation and library code.

We also now have an assembly language file which has been assembled into PIC Hex code for programming into a real device.

5.3 The hex file – programming a real PIC

Standard PIC assembler code is created by WIZ-C Professional, it is created in the output directory of your project. Here there will now be a hex file called "First.hex", this could be used directly to program a PIC16F84 and test it. If you have an FED PIC Programmer then you could use the **Tools | PICProg** menu option which will start the programmer with the correct hex file. You can also configure the tools menu to add other programmer types, or use MPLAB to program the hex file. If you have a different type of programmer then consult the manual to determine how to load a hex file and select the file "First.hex" which can be found in the First\Output directory. Then follow the instructions to program your PIC. Fuses should

be selected as XT, no watchdog timer and Power up Timer enabled. Connect an LED via a 300R resistor to Port B, Bit 0 (pin 6 of the PIC) and power up, it should flash rapidly.

Whether you have a demonstration circuit or not we can simulate the file within the C Compiler but first we'll look line by line at this program.

5.4 Program description - line by line

This first program is very simple and we will look in detail at the structure of a C program later in the manual, however for the present we will examine each line in detail before simulating the program. To describe the program, it is repeated a number of times with certain lines being highlighted - the description for the highlighted lines is described below the program.

```
//
// Example program - LED Flasher
//
#include <pic.h>           // Includes all relevant PIC definitions
#include <delays.h>

void main()
{
    TRISB=0xfe;
    while(1)
    {
        Wait(250);
        PORTB=PORTB^1;
    };
}
```

These 3 lines are comments - in ANSI C a comment may start with //, anything on the line following the // will be ignored.

```
//
// Example program - LED Flasher
//
#include <pic.h>           // Includes all relevant PIC definitions
#include <delays.h>

void main()
{
    TRISB=0xfe;
    while(1)
    {
        Wait(250);
        PORTB=PORTB^1;
    };
}
```

These two lines include other files. In the case of our program the PIC to be used is the PIC16F84 and the first line includes information which is essential for the compiler, and which tells it which variables are needed for that PIC. The file pic.h automatically includes the header file for the processor which has been selected for the project. In this case it will load the file P16F84.h which is included within the libs sub-directory of the WIZ-C Professional installation - you can open this within the editor to see what it contains (simply use **File | Open**).

The files pic.h and delays.h are called header files. The definitions of the Ports and other 16F84 special function registers are included in the first header file pic.h.

Our program uses a library *function*. A function is a collection of code (similar to a sub-routine in BASIC) which may be called from anywhere in the main program. In this case our library function is Wait (which is used further down in the program), however the compiler needs to be told about Wait before we try to use it. The header file - delays.h - includes a number of definitions of library functions including the Wait function.

```
//
// Example program - LED Flasher
//
#include <pic.h>           // Includes all relevant PIC definitions
#include <delays.h>

void main()
{
    TRISB=0xfe;
    while(1)
    {
        Wait(250);
        PORTB=PORTB^1;
    };
}
```

Statements in C are grouped into blocks of codes called functions which we will look at in detail in later chapters. In this case the function is called main. The void keyword simply states that main does not return a value. The curly brackets surround all the code within the function called main. In C there is *always* a minimum of one function, and there is *always* a function called main. main is called first, so the first line of code in main is the very first code executed in the program. Even using the application designer (which automatically creates a lot of code for us) still creates a main function.

```
//
// Example program - LED Flasher
//
#include <pic.h>           // Includes all relevant PIC definitions
#include <delays.h>

void main()
{
    TRISB=0xfe;
    while(1)
    {
        Wait(250);
        PORTB=PORTB^1;
    };
}
```

This line shows how to set one of the tri-state registers - in this case the register for Port B. All of the pins of Port B are to be inputs except pin RB0 - bit 0 of Port B - which is to be an output. Values in C normally taken to be decimal. In this case the '0x' shows that it is a hex number.

```
//
// Example program - LED Flasher
//
#include <pic.h>           // Includes all relevant PIC definitions
#include <delays.h>

void main()
{
    TRISB=0xfe;
    while(1)
    {
        Wait(250);
        PORTB=PORTB^1;
    };
}
```

This is a loop. The code in the loop will be repeated whilst the condition is true (non-zero), in this case the code in the loop will repeat forever. The curly brackets surround a block of code and show which code is repeated.

```
//
// Example program - LED Flasher
//
#include <pic.h>           // Includes all relevant PIC definitions
#include <delays.h>
```

```

void main( )
{
    TRISB=0xfe;
    while(1)
    {
        Wait(250);
        PORTB=PORTB^1;
    };
}

```

This is the library function which delays a period of time. In the Editor window click the word Wait, and press Ctrl and F1 together which will bring up a help file entry for this function. This function needs to be told how long to wait - the number in brackets is the delay time which is in milli-seconds. So this causes the program to stop for 250mS – note that the Wait function is approximate, accurate timing would use one of the PIC's internal timers. In practice in this example the Wait function is accurate to 0.2%.

```

//
// Example program - LED Flasher
//
#include <pic.h>           // Includes all relevant PIC definitions
#include <delays.h>

void main( )
{
    TRISB=0xfe;
    while(1)
    {
        Wait(250);
        PORTB=PORTB^1;
    };
}

```

This is the final line of the program loop. The ^ symbol is the C operator for a bitwise exclusive OR function. Port B is exclusive OR'd with the number 1, this will cause bit 0 of Port B to be inverted, so every time that the loop executes Port B, bit 0 will change state causing the LED to flash.

5.5 Initial simulation

When the program is first compiled the program counter is reset and the edit window will show the initial word of the program in the first assembly file. The simulator will run past all of the C initialisation code which clears memory and sets up the system and then leave the program at the first line in the program which will be highlighted with a blue bar.

5.5.1 Switching screen layouts

There is a large amount of information provided on the screen and to aid users there are 3 main views :

Compact	Press ALT+C keys
Debugging	Press ALT+D keys
Editting	Press ALT+E keys

In compact mode all windows (Debug, Project, Editting and Information) are shown on screen. In Editting mode the debug window is hidden and most screen space is given to the edit window. In Debugging mode most space is given to the debug window.

FED recommend that users get used to using the ALT and C, D or E keys to rapidly switch views. Normally only the ALT+D and ALT+E modes will be use, the compact mode is provided for existing users and is similar to previous versions of our environments.

The time will be shown at about 320uS which is how long the C program takes for initialisation of memory.

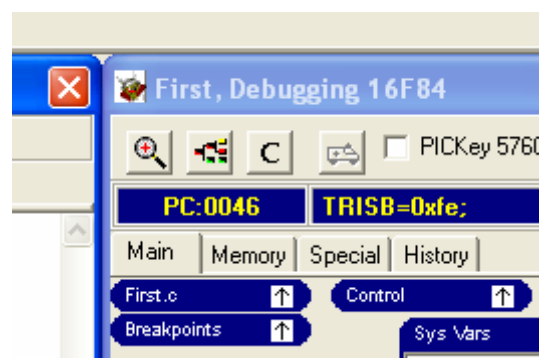
First we'll take a look at the screen elements.

5.5.2 Screen elements during debugging

5.5.2.1 Debugging Window

The debugging window shows a large amount of information about the processor. We'll just take a look at some of the key features.

The current hex value of the PC and the source line of the program is shown at the top of the debugging Window : Here we can see the first line of the program shown and the PC set to 46 hex.

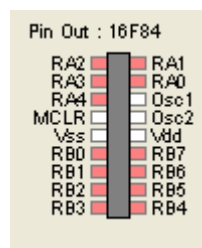


The Sys Vars view on the debugging window shows the value of internal registers and also the PIC ports :

Sys Vars	
Name	Value
Status (SREG)	1C [- - - T P ...
W	61 3DH
FSR	04FH -> 00 0...
PORTA	1F
PORTB	FF
sp	4Fh
0x040	00 00 00 00 ...
ACC	00

Of most interest here is the value of PORTA and PORTB.

The pinout shows the current state of the pins of the PIC :



The pin states are as follows :

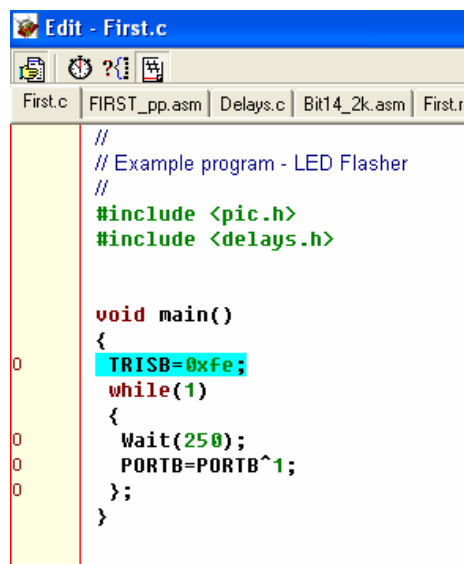
Colour	State
Light Red	Pin is an Input and is high (Level 1)
Dark Red	Pin is driving as an output and is high (Level 1)
Light Green	Pin is an Input and is low (Level 0)
Dark Green	Pin is driving as an output and is low (Level 0)
Blue	Pin is an analogue input (not applicable to 16F84)
White	Pin is not an input or output, e.g. Power, Oscillator etc.

Watch the colours on PORT B bit 0 as we simulate the device.

The Variables window on the debugging window is currently blank, you can use it to view the value of C variables as we simulate the program.

5.5.2.2 Edit Window

During simulation the edit window shows much useful information :





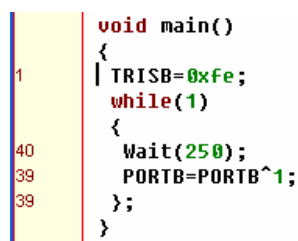
```

//
// Example program - LED Flasher
//
#include <pic.h>
#include <delays.h>

void main()
{
    TRISB=0xfe;
    while(1)
    {
        Wait(250);
        PORTB=PORTB^1;
    };
}

```

Here we can see the current execution line of the program which is the first line of main(). This is the blue highlight. The bar at the left shows information about each line as we execute. By default it shows the number of times each line has executed. It can also show the address of each line (use the  button at the top of the window), or it can show the time at which each line was last executed (use the  button). Here is the window after the program has run for 10 seconds :



```

void main()
{
    TRISB=0xfe;
    while(1)
    {
        Wait(250);
        PORTB=PORTB^1;
    };
}

```

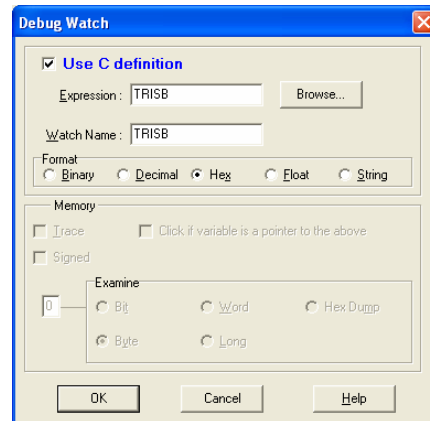
5.5.3 Test first line

Now press F7 to step past the first line.

At this point one C command has been executed :

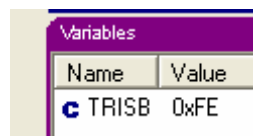
```
TRISB=0xfe;
```

Look at the debugging window. We would also like to look at the tristate register for PORTB to ensure that it has been set correctly. To check add the TRISB register to the debugging window. Use the **Simulate | Watch | Add Watch** menu option (or click the Variables window on the debugging window and press the insert key).



Either type in TRISB, or click on the browse button and select the TRISB line from the list box, and click on the >> button. Press the OK button to add this file variable to the list. Note we have a number of options here. The Use C Definition box default will search the C source for the variable, unchecking this will allow internal PIC registers to be inspected. You can also view in Binary, Hex or Decimal etc.

Check that TRISB holds the value 0xfe :



5.5.3.1 Check delay and port B toggling

Now we want to check the delay routine, but we don't want to step into the delay routine. In addition we want to simulate at full speed to avoid waiting for the program to run through the delay routine. At the top of the main window is a slider called Update Rate. Move this to the extreme right, and then click it left one position :




This will simulate at maximum speed whilst still updating the debugging window.

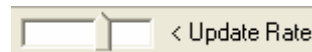
Press F8 - the command to step over the next line in a C program. This may take a couple of seconds and when the program stops check the time shown at the top of the screen. It should be around 250mS - the Wait function is not intended for very accurate timing functions :

250.875mS

Look at PORTB and check its value (it may be FF or FE as the port is not initialised by the C Compiler). Press F7 and watch it change as the final line of the loop which toggles Bit 0 of the port is executed. We will now be back at the bottom of the loop.

5.5.3.2 Reset and run the program at full rate

Now reset the processor - Use the **Simulate | Reset Processor** menu option (or press the blue button on the tool bar : ). Now press F9 - this will run the program at full rate, watch the PORTB variable and you should be able to see it switching from FE to FF and back again. If you can't then slow the program down by sliding the update rate down :

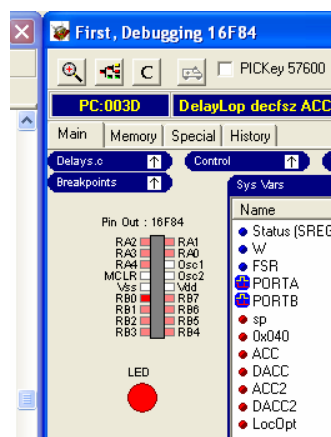


Run the simulation until the time shows a few seconds and then click the Stop Simulation button .

5.5.3.3 Using an LED to examine the output

WIZ-C has the capability to simulate LED's, switches, LCD displays and a number of other devices which might be connected to the PIC.

We'll start with the LED. Use the *Simulate | Add External Device* menu option. A dialog box will come up. In the External Device type box select LED. There are a number of parameters and values which may be selected for each device. For the LED most of these can be ignored apart from the connections. Under the Connections box there is a list box called Pins with two entries "Anode" and "Cathode". Select Anode and then use the Port box to select Port B, use the Bit box to select bit 0. Select Cathode and click the Connect Low option. This will connect the LED between Bit B0 and ground. Press the OK button. Note how the LED appears on the debugging window. You can drag the LED by clicking just above it and dragging it on the window. We moved it like this :

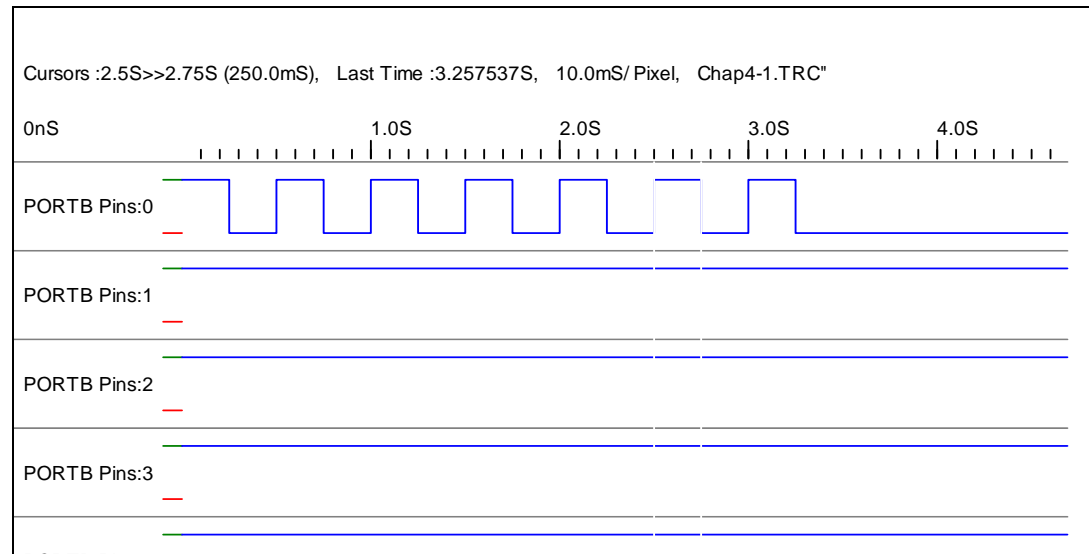


Note that the LED is illuminated. This is because the simulator assumes all unconnected inputs to be logic 1. Run the program again and watch the LED flashing – again you might need to reduce the update rate to see it.

5.5.3.4 Waveform window

Finally we will examine the simulation on the Waveform analyser - this will display a graphical view of the output. Use the **Tools | Examine Wave Window** menu option. A dialog box will appear "Define Trace Format". Click the trace name box and select "PORTB Pins", click the "Add as 8 Line Traces" button. This will show all pins of Port B, but the time scale will be so short that nothing will appear to be happening. Press F8 to zoom in - and then press it again

and again and again until you can see Port B bit 0 toggling (I needed to press it 16 times to get a good view on the output). Check the timescales, it should be changing every 250mS or so - you can measure the time between events by dragging the vertical black lines at either end of the window and looking at the time difference in the Cursors box at the top of the window.



5.6 Program for the 16F877 development board

If you have the FED 16F877 development board you can run a similar program to flash one of the LED's on the board. The LED's on this board are multiplexed with a number of other functions. The four LED's are connected to pins D4 to D7, they are all turned on with a transistor drive on pin E2, when pin E2 is high the LED cathodes are connected to ground using an NPN transistor (consult the circuit in Chapter 2).

This program is entered in exactly the same way - create a new project and enter this code:

```
//
// Example program - LED Flasher
//
#include <pic.h>
#include <delays.h>

#define __config __CP_OFF & __BODEN_ON & __WDT_OFF & __PWRTE_ON & __HS_OSC & __LVP_OFF

void main()
{
    ADCON1=7;
    TRISD=0xef;
    TRISE=3;
    PORTE=4;
    PORTD=0;

    while(1)
    {
        Wait(250);
        PORTD=PORTD^0x10;
    };
}
```

Note this has a number of differences from the last project. The LED's on the development board are on bits 4,5,6 and 7 of Port D, so the tri-state register for PORTD needs to be set to drive bit 4. Similarly the XOR value is now 10 hex to toggle bit 4. The other lines are described below:

```
#__config __CP_OFF & __BODEN_ON & __WDT_OFF & __PWRTE_ON & __HS_OSC & __LVP_OFF
```

In summary these lines define :

- No Code Protection
- Brown Out detection On
- Watch Dog timer off
- Power Up timer On
- HS Oscillator type
- Low Voltage Programming pin off

This line defines the configuration fuses for the device which are stored in the hex file and thus do not need to be defined in the programmer. In this case we have manually assigned them, this line can be deleted and the **Project | Set Configuration Fuses** menu option used instead.

```
ADCON1=7;
```

The 16F877 has internal Analogue to Digital converters which operate on pins RA0,RA1,RA2,RA3,RA5,RE0,RE1,RE2. When the chip is reset all of these pins are configured to act as analogue inputs and cannot be used for digital inputs or outputs. In our program we wish to make use of pin RE2. This line sets all of the A/D pins to be digital inputs or outputs, it should be included in any program you write where the A/D converter is not required.

It is possible to change the number of A/D inputs and digital I/O's. We will look at this later when looking at operation of the A/D converter.

```
TRISE=3;  
PORTE=4;
```

These two lines set port E to drive bit 2, and then bit 2 (pin RE2) is set high to turn on the transistor to drive the LED's. Note that port E only has 3 external pins and therefore TRISE is set to 3 to drive pin RE2 and not to FB hex as might be expected.

```
PORTD=0;
```

This initialises port D so that it starts at 0 - this is not strictly necessary, but is useful for exercise 2 at the end of the chapter.

This completes the tutorial.

5.6.1 Compile and simulate

Use the **Project | Current Project Options** menu option to change the project to use a 16F877 processor and set the processor frequency to 20000000.

Press F9 to simulate in the same way as before - note the processor simulation will be much slower as the simulation will be running 5 times as many instructions with the higher clock frequency. Watch PORTD as it changes from EF to FF, stop the simulation and reset the processor.

Note also that the library function Wait is still waiting 250mS even though the clock frequency has increased 5 times, this is because the function adjusts automatically to the project clock frequency.

5.6.2 Simulating with external devices

We'll start with the LED. Use the *Simulate | Add External Device* menu option. A dialog box will come up. In the External Device type box select LED. There are a number of parameters

and values which may be selected for each device. For the LED most of these can be ignored apart from the connections. Under the Connections box there is a list box called Pins with two entries "Anode" and "Cathode". Select Anode and then use the Port drop down box to select Port D, use the Bit drop down box to select bit 4. Select Cathode and click PORTE bit 2. Note that the cathode of the four LED's is connected to PORT E bit 2 by a transistor which inverts the sense of the data drive, so in the parameters box where the option is given for transistor drive, select Yes.



This will connect the LED between Bit B4 and ground. Press the OK button. Note how the LED appears on the debugging window.

Run the program and watch the LED flash

5.6.3 Program the development board

The development board manual shows how to program the board. Use the programmer to blow the chip, not that the configuration fuses are defined in the main C file and do not need to be defined again in the programmer.

This completes the tutorial.

5.7 Exercises

1. Can you rewrite the 16F877 development board version to flash all 4 LED's instead of just one ? Add the LED's to the development window.
2. Now re-write so that LED's 1 and 2 flash out of time with LED's 3 and 4.
3. Re-compile the 16F877 version with Processor Frequency of 4MHz (4000000). The board has a 20MHz clock - what do you think will happen when the program is run on the wrong processor frequency? Try it out.

6 Comments and statements

Before we look in detail at how the C language works, we'll take a look at the fundamentals of the program structure.

6.1 Comments

You can include comments anywhere within the C program.

The first type of comment is a comment on a line which begins with a double forward slash `//`. Anything on a line following the `//` is ignored, however all following lines are fine:

```
// This is a comment on a line of its own

    // This is a comment on a line of its own, but indented

char x;    // This is a comment following some C Code
```

The second type of comment is a grouped comment. `/*` starts a group comment, and all information following the `/*` is ignored until a `*/` is encountered. This also includes any line breaks between the beginning and end of the comment.

```
/* This
is a group comment
on a number
of lines */

/* this is a comment on a line of its own */

char x;    /* This is a comment following some C Code */
```

In practice the group comment is a little old fashioned and is mainly used for blocking out large chunks of code for debugging purposes.

6.2 Statements

Most C Code is made up of statements. A statement may be an expression, or a call to a function, or a declaration of variables or subroutines (functions). A simple statement is followed by a semi-colon, and may break over several lines:

```
x=y*z+a*b;           // A statement

x=y*z
+
a*b;                 // Statement on 3 lines is identical in effect
```

A number of statements can be grouped in a block which starts and ends with curly brackets, this is called a compound statement. For example

```
if (x>0)    // Test
{
    // Compound statement
    x=1;
    y=2;
}
```

7 Variables and Constants

7.1 Introduction

In this chapter we shall look at the way in which C holds variables and constants. There is an introduction, and then we'll take a look at how variables are used within WIZ-C PROFESSIONAL.

Use the project VAR.PC which can be found in the Projects\Learn To Use C\07 - Var directory within the WIZ-C Professional installation.

7.2 Constants within C

There are two types of constant in C. Those which are simply numbers, strings, (or constant arrays), and those which are symbolic - where a name represents a constant value. We'll look at symbolic constants later, here we'll look at constants within a programme.

7.2.1.1 Numbers

The default base for all numbers in C is decimal, so any number written without a prefix is taken simply as a decimal representation. This is how you might write the value 65:

```
65
```

To write a hex value, then it should be preceded with a 0x. Here is 65 in hex:

```
0x41
```

Octal values can be written in C, but are rarely used, they can also be very confusing. An octal value is taken when the first digit is 0. Here is 65 in octal:

```
0101
```

Characters can also be used in place of numbers. 65 in decimal is ASCII for the letter A. To enter a letter as its ASCII number then surround the letter with single quotes. Here is 65 as a letter:

```
'A'
```

7.2.1.2 Strings

Strings in C are simply represented by surrounding the wanted string in double quotes ("). Here is a string:

```
"String"
```

It is also important to note that if two strings are written together then they are concatenated as if they were one string. This enables long strings to be written on several lines.

For example this :

```
"A long" " String"
```

is equivalent to this string:

```
"A long String"
```

and also to these two lines :

```
"A long"
" String"
```

Strings in C are actually character arrays - we'll look at these in more detail in the section on arrays.

7.2.1.3 Special characters

It is possible to enter special characters - either within a string, or in the special quote form. This is handled by preceding the special character with a backslash. For example:

```
"String\non two lines"
```

The special characters \n represents the number 10, which is a line break. Similarly:

```
'\n'
```

is actually the same as the number 10.

Here is the total list of special characters:

```
\digits      digits are number from 0 to 7 *
\a
\b
\f
\n           Line break - value 10
\r           Carriage return - value 13
\t           tab - value 8
\v
\x           Followed by a hex number **

* - The \ and all digits are replaced by the octal number
**- The \x and all hex digits are replaced by the hex number
```

7.3 Declaring variables

In C it is necessary to tell the compiler about all the variables which are available before using them. This enables the compiler to know what sort of variable it is, and how much space it takes. This process is called declaring variables.

There are two main types of variable - global and local. We'll take a look at local variables in the chapter on functions, here we'll look at global variables.

Global variables are those which are available anywhere in the program - that is that once they have been declared then any part of the program can read (or in most cases write) to the variable. The compiler allocates space for global variables within the memory of the processor.

To declare a global variable the type is given followed by the variable name. Here is an example of how to declare two global variables - x and y :

```
char x;
char y;
```

The char keyword is the type of the variable - there are a number of types which we'll look at in the next section. A char type is actually 8 bits, and can store numbers from -128 to +127, it takes up one File Register in the PIC, so the compiler will allocate space for x and for y as two single byte variables in RAM.

To declare any variable a statement is written with one or more type keywords (char is a type keyword) followed by the variable name, followed by a semi-colon. Several variables can be declared at once by separating their names with commas:

As x and y are the same type they could also be declared as follows:

```
char x,y;
```

During this chapter we'll build up an example program which will look at various types. To start with open a new project called "Var" with a single C file called "Var.c", enter the following code:

```
//
// Looking at variables
//

char x,y;

void main()
{
endit:
while(1);
}
```

In this example program we have declared two character variables x, and y. There is only one line in the main program `while(1);`. This is an infinite loop which we'll use to break the program when it is finished. The label `endit:` defines a line in C which can be the destination of a jump, we'll use it to insert a breakpoint.

7.3.1 Adding a breakpoint

Now we'll add a breakpoint to the program. Use the **Simulate | Add Breakpoint** menu option to bring up the breakpoint dialog. We would like to add a breakpoint at the label `endit:`. However C creates a label within the assembler which is made up from the function name and the label name. The label has the form:

`FunctionName_LabelName`

So our label has the assembler name `main_endit` as the function is called `main`. So enter the label `main_endit` into the address box and click OK. Move the update rate tab as before :



Press F9 to run the program and watch how it hits the breakpoint at the moment nothing exciting will have happened so we'll leave it there for the moment..

This is the method that we will use to stop any of the example programs after they have finished an operation that we would like to examine.

You will notice that there is a little blue circle by the line `while(1)` :

```
endit:
  while(1);
}
```

The blue circle denotes a breakpoint. If you click it you can turn it gray – to disable it, click again to turn it off, or finally click the point where the circle was to turn it on again – this is very useful as a method for rapid setting of breakpoints and much quicker than entering the label value, so in future we'll use this method.

7.4 Variable Types

7.4.1 Fundamental Types

There are a number of types built in to C, and it is possible to define your own types - we'll look at this later. In this section we'll take a look at the built in types.

7.4.1.1 char

The char type is used for 8 bit values which can vary from -127 to +128. Within WIZ-C Professional the char type is the second most efficient type for variables. Use the char type whenever a number is required which can fit within this space.

7.4.1.2 int and short

The int and short types are used for values which are defined by the compiler. This is one of the confusing areas of C. For example C Compilers for Windows may define int as being used for 16 bit values, but some compilers define int as being used for 32 bit values.

For the WIZ-C Professional compiler the int and short types are identical. They both define a type which is 16 bits long, and can store numbers from -32768 to +32767. On the PIC arithmetic using int and short values is much slower than char, and so values should be defined as char if they can fit within the char type.

7.4.1.3 long

In similar fashion to int values, long values are defined by the compiler.

For WIZ-C Professional a long value is defined as being 32 bits. Therefore it can store values from -2147483648 to +2147483647. Long arithmetic on the PIC is slower again than int, and so long values should be reserved for greatest precision.

7.4.1.4 float

The **float** type is a 32 bit type which can hold floating point numbers. The float form consists of a sign bit, 8 bit exponent and 24 bit mantissa. The top bit of the mantissa is always 1, and is not saved in the number. Float values can represent results positive or negative numbers within the approximate range 1e-38 to 1e+38.

7.4.1.5 Unsigned

If the keyword unsigned is used before any of the integer types shown above then it makes the value unsigned. For example:

```
unsigned char x;
unsigned long y;
```

Within WIZ-C Professional the unsigned char type is the most efficient way of storing a value - it can store values from 0 to 255 inclusive. It is important to differentiate between signed and unsigned values. Consider the following code:

```
unsigned char x;
char y;

void main()
{
    x=0xf0;
    y=0xf0;
    x=x/2;           // Divide x by 2
```

```

    y=y/2;           // Divide y by 2
endit:
    while(1);
}

```

Note that both x and y are initialised to the same value - 0xf0. However after this code is run, x will have the value 0x79 (120 in decimal), but y will have the value 0xF8 (-8 in decimal).

7.4.1.6 Summary of types in WIZ-C PROFESSIONAL

Here is a table showing all the types that we have looked at so far:

Type	Example	Number of bytes required for a variable	Minimum Value	Maximum Value
char	char x;	1	-128	+127
unsigned char	unsigned char x;	1	0	+255
int	int x;	2	-32768	+32767
unsigned int	Unsigned int x;	2	0	65535
long	long x;	4	-2147483648	+2147483647
short	short x;	2	-32767	+32768
unsigned short	Unsigned short x;	2	0	+65535
float	float f;	4	~ -1e38	~ 1e38

7.4.2 A closer look at watch variables.

Now we can develop a program looking at all the types and examine them in the watch window. Change the vars program so that the main program reads:

```

unsigned char x;
char y;
unsigned int i;
int j;
unsigned long l;
long m;

void main()
{
    y=0xff;
    x=y;
    i=y;
    j=y;
    l=y;
    m=y;

endit:
    while(1);
}

```

Now compile the program. Click the debugging window, make sure the Watch tab is foremost and press insert.

To add x, press insert, enter the value x into the Expression box, click byte, click decimal, make sure the Signed box is clear and press OK.

To add y, press insert, enter the value y into the Expression box, click byte, click decimal, make sure the Signed box is checked and press OK.

This will add the variables x and y and display them so that x is shown as an unsigned, 8 bit value, and y is shown as a signed 8 bit value.

Repeat with i and j, only this time make sure that the Word option is selected. Finally add l and m and make sure that the Long option is selected.

Now reset the processor by clicking the blue button with the white 0. Note that all the values shown display zero. This is because the compiler (and all other ANSI C compilers) initialises all variables to 0 when it starts running.

Press F7 again and again to single step the program - watch how the variables are all set to the value of y. Also note how the compiler automatically extends the sign of y into the upper bits of longer variables. When the program completes the following values will be shown.

• x	M	255
• y	M	-1
• i	M	65535
• j	M	-1
• l	M	-1
• m	M	-1

Change the initial value of y to 7 and run again. This time the upper bytes of the longer variables will be set to 0.

7.5 Arrays & Strings

An array is a collection of variables, all of the same type under a single name. An array is declared with a type, the name, and then square brackets, [], enclosing the number of variables collected in the array:

```
char x[10];
```

This declares an array called x, which contains 10 variables which are all type char. As char types use one byte of RAM, then the array occupies 10 bytes of RAM. The first variable in the array x is referred to as x[0], the second as x[1] and the last variable in the array is x[9].

We can refer to any one of the variables which make up x by using the square brackets:

```
x[0]=1;
x[1]=2;
y=x[0]+x[1];
```

In this example the first variable in array x is set to 1 and the second to 2. The variable y is set to the value of these two items within array x added together and will be 3.

Delete all the lines of your program which have been used for previous examples and add this example to your program. It should now look like this:

```
unsigned char x[10];
char y;

void main()
{
    x[0]=1;
    x[1]=2;
    y=x[0]+x[1];

    endit:
    while(1);
}
```

Now compile your program and run it (press F9). Check that y has the value 3. Now if you look in the debugging window you will find that x is shown, but is only displayed as the first value in the array. Click x in the watch window and press the Enter key to change the way it is displayed. Now select Hex Dump. The watch window will now shown 16 values from memory starting with the first address of x. Note how x[0] and x[1] have been set to 1 and 2 respectively.

It is possible to use a variable to refer to the variables in an array. $x[n]$ will be the first item in the array if n is 0, and the 5th item in the array if n is 4. Look at the following program, this achieves the same result as the example, but uses variables to "look up" values in the array x .

```
unsigned char x[10];
char i,j,y;

void main()
{
    i=0;
    j=1;

    x[i]=1;
    x[j]=2;
    y=x[i]+x[j];

    endit:
    while(1);
}
```

Arrays can have more than one dimension (in this case it is an "array of arrays"):

```
char x[2][3];
```

In this case x is an array of 2 arrays, each of which holds 3 characters. The very first item in the array is $x[0][0]$, the second is $x[0][1]$, the 4th item is $x[1][0]$, the last item in the array is $x[1][2]$. The total size of this array is 6 characters, or 6 bytes. Again variables can be used instead of numbers.

A string is used in C as if it is an array of characters, however following the last character there is a 0 character. For example :

```
"ABC"
```

is held in memory as the bytes 41 hex, 42 hex, 43 hex and then 0 hex which represents the end of the string. Therefore this string takes 4 bytes in memory. This is very important - as we'll find later when manually making space for strings it is vital to remember that however long a string appears to be, it will always take an additional byte in memory for a 0 at the end of the string. This must be the most common error ever made by C programmers !

7.6 Constant declarations

So far we have looked at variables. However C lets us define constant values which are very useful for defining ports or bits or values which are characteristic of our system, but which can be changed - for example when changing hardware.

To define a constant then simply place a `const` keyword in front of the value, and follow with an equals sign and the value:

```
const unsigned char adPORTA=5;
const char Command='A';
const long NotFound=-1;
```

Now you can use these values in any expression (equation), and they will behave as though the number were being used.

7.7 Enumerated constants

Enumerated constants are a method within C of automatically assigning values to constants in sequence. As an example of the syntax :

```
enum {Start,Stop,Pause,FF,REW};
```

This defines 5 constants. Start takes the value 0, Stop has the value 1, Pause has the value 2, FF - 3, and REW 4. Each constant takes the value before and adds one to it.

You can use an equals sign in an enumerated constant:

```
enum {Start,Stop=8,Pause,FF,REW};
```

Now Start has the value 0, Stop has the value 8, Pause has the value 9 etc.

You can use an equals sign for every value if you like:

```
enum {Start=1,Stop=2,Pause=4,FF=8,REW=16};
```

The constants take the values shown.

A constant which is defined by enum will be of type char if all the values in the enum could fit within a char variable (that is every value is between -127 and +128), and int if they could fit in an int variable.

The final very useful feature of enum statements is to define a new type:

```
enum controls { Start=1,Stop=2,Pause=4,FF=8,REW=16};
```

Now you can use `controls` as a keyword whenever you are referring to an item which holds one of these values. In this case `controls` is equivalent to `char`. For example:

```
controls Button;
```

7.8 User defined types

It is possible to define new type keywords. As we will see this is most useful when we are dealing with structures and unions, however it can be used to shorten type definitions. To define a new type the keyword `typedef` is used, followed by the type, followed by the new type name. For example if we want to define a new type called `BYTE` which is equivalent to an `unsigned char` type, then we could use:

```
typedef unsigned char BYTE;
```

In fact this is defined in all of the WIZ-C Professional header files. Whenever you use a statement such as:

```
#include <pic.h>
```

then the type `BYTE` will be defined as shown here. Now you can use `BYTE` like any of the built in types:

```
BYTE x;
BYTE y[10];
```

7.9 Initialisation

All variables can be initialised when they are declared with an equals sign:

```
char x=7,y=8,z;
```

This defines `x`, `y` and `z` as character variables. Before the program starts in the main function the value of `x` will be set to 7, `y` to 8 and `z` will be left at 0 (recall that all variables in memory are initialised to zero).

We can also initialise the values in an array, this is done with curly brackets enclosing the complete set of values which are separated by commas:

```
char x[5]={0,1,2,3,4};
```

This sets x[0] to 0, x[1] to 1 etc.

Try this program - examine variable x as a hex dump. (You will need to run the program with F9).

```
//
// Looking at variables
//

char x[5]={0,1,2,3,4};

void main()
{

endit:
while(1);
}
```

To initialise an array of 2 or more dimensions then groups of bracketed lists can be used:

```
//
// Looking at variables
//

char x[2][3]={{9,8,7},{6,5,4}};

void main()
{

endit:
while(1);
}
```

Use this program in WIZ-C Professional and again look at how the values are stored in memory for the array x.

The compiler is capable of calculating array size from the initialisation data. For example:

```
char x[]={-1,0,1};
```

In this case x is declared to be a 3 character array and the values are initialised in order to -1, 0 and 1.

Finally it is possible to initialise arrays with strings:

```
char x[]="123",y[]="abc";
```

Try this program:

```
//
// Looking at variables
//

char x[]="123",y[]="abc";

void main()
{

endit:
while(1);
}
```

Now examine the memory from address x as a hex dump - notice how the string "123" is represented in memory by the ASCII codes 0x30,0x31,0x32 (or in decimal 48,49,50), however note also that there is a 0 byte after the string, and then you can see the 4 bytes which make up the string called y. Both x and y are arrays of `char` types, but have been initialised with a string.

7.10 Casting

It is possible to convert types during a calculation - we won't make too much of this at present as it is far more useful when considering pointers. To convert a value from one type to another then precede it with the wanted type in brackets - this example casts x to from char to unsigned char :

```
char x;
int y;
...
...
y=(unsigned char)x;
```

To see how casting operates try this program and look at the variables x and y when the program has run:

```
//
// Looking at variables
//
char x;
int i,j;

void main()
{
    x=-1;
    i=x;
    j=(unsigned char)x;
endit:
    while(1);
}
```

Examine i and j in the Watch tab of the debugging window.

• x	M	-1
• i	M	-1
• j	M	255

At the end of the program i will be 0xffff (-1 in decimal) and j will be 0xff. The difference is owing to the cast of x to unsigned char in the expression assigning it to j.

7.11 An important note on casting

By default WIZ-C will cast up to the longest type, so if you undertake an addition between a long type and a character type the character type will be converted to a long before the addition. However if all types are the same then they will be left at the current type length.

This can lead to unexpected results if you want a longer type from two shorter types. For example consider combining the ADRESL and ADRESH registers which store the results of an Analogue to Digital conversion into a single 10 bit value:

```
int volts;

volts=ADRESL+(ADRESH<<8);
```

Now if ADRESL was 0x10 and ADRESH was 0x2 you might expect volts to hold the value 0x210 after the expression. In fact as all the types on the right hand side of the expression are taken as unsigned character the result will simply be 0x10 – the value of ADRESH has been shifted out of a character and is lost. The answer in this case is casting :

```
int volts;

volts=(int)ADRESL+((int)ADRESH<<8);    // Works fine
```

This doesn't cause an issue very often but programmers do need to be aware of it.

7.12 External variables

Whenever a variable is declared then the compiler allocates space for it. Therefore if a variable is declared twice within the same program then an error will be created as the compiler allocates space for it twice. To allow for this it is possible to precede the declaration with the `extern` keyword which declares the variable but does not allocate space for it. This is how to declare a variable as external:

```
extern char x;
```

This allows for more than one C file within a project. In the first file the variable is declared, in the second file it is declared, but no space is defined for it:

1st File:

```
char x;
```

2nd File:

```
extern char x;
```

We'll look at the uses of `extern` and how it can be used in the major project which completes this manual.

7.13 Exercises

1. Declare an array of type `int` with 4 variables. Now set the values of the array to any numbers you like. Examine the array using a hex dump - can you work out how WIZ-C Professional stores integers in memory ? Repeat with a long array.
2. Try some of the types and keywords shown above that we haven't yet seen as examples - do they do what you expect ?

8 Expressions

8.1 Operators

So far we've looked at how to declare variables. C has a wide variety of operators for expressions, some of which will be familiar and some of which may not. C has a priority for operators - higher priority operators are performed before lower priority. Try this program:

```
char x,y,r;

void main()
{
    x=5;
    y=8;
    r=y+3*x;
endit:
    while(1);
}
```

z will end up as 23 as the multiply operation (*) has higher priority than the addition operator (+). (If there were no priority then z would be 55).

In this section we'll look at all of the operators available in C - however some of them are specific to features of C which we haven't yet looked at and so we'll cover those at a very high level.

Use the project Expressions.PC which can be found in the Projects\Learn To Use C\08 - Expressions directory within the WIZ-C Professional installation.

8.1.1 Logical Results

In C there are a number of operators which give logical results - for example `a>b`. These operators return a 1 for true, or a 0 for false. C always takes a 0 value to be false and a non-zero value to be true.

8.1.2 lvalues

In C an lvalue is an expression which may be assigned to - it is an item or an expression which may appear on the left hand side of an = sign. The following are all lvalues (note that we will cover pointers in a later chapter):

```
x           // x is a variable
x[7]        // x is a variable array
*xp         // xp is a pointer to a character
```

All of these examples can appear on the left hand side of an expression:

```
x=y;
x[7]=x[1];
*xp=9;
```

8.1.3 () []

Bracketed operations always have the highest priority. () brackets group operations which must be performed before others, [] brackets are for arrays.

8.1.4 -> .

Member operations for structures and unions. -> is used to determine a member of a structure or union from a pointer and . is used to determine a member of a structure or union.

```

struct Time
{
    unsigned char Mins;
    unsigned char Hours;
};

void main()
{
    Time Now;
    Time *tp;

    tp=&Now;

    tp->Mins=0;
    tp->Hours=12;

    Now.Mins=Now.Mins+2;
endit:
    while(1);
}

```

8.1.5 !

This is a logical operator. If the argument is true then it returns false, if the argument is false (zero) it returns true.

```

X=!7;
Y=!0

```

After these statements X will be 0 and Y will be 1.

8.1.6 ~

This inverts all the bits of its argument

```

char x=0xaa;
x=~x;

```

After these statements x will be 0x55 (all the bits set to one are set to zero and vice versa).

8.1.7 + - (Unary)

When a + or a - sign is placed in front of a number or variable it has the effect of leaving it as is (+), or negating it (-).

```

char x;
x=-1;

```

8.1.8 ++ --

These operators may be unfamiliar to users of other languages. They only operate with lvalues (expressions or items which can appear on the left hand side of an equals sign). ++ adds one to the item, -- subtracts one from the item.

If the ++ appears before the item then one is added and the result is the new value, if the ++ appears after the item then one is added and the result is the old value (before the one was added). An example may help to make this clearer (you can run this in the simulator as usual)

```

char x,y;

void main()
{
    x=1;
    y=++x;    // After this y is 2 and x is 2
    y=x++;    // After this y is 2 and x is 3
}

```

```

y=x++;    // After this y is 3 and x is 4
y--;      // After this y is 2
--y;      // After this y is 1
x=y--;    // After this x is 1 and y is 0

endit:
while(1);
}

```

8.1.9 &

We haven't yet looked at pointers and addresses. However the & operator returns the address of an lvalue.

```

char x,*xp;
xp=&x;
*xp=9;           // Sets x to 9

```

8.1.10 *

We haven't yet looked at pointers and addresses. However * dereferences a pointer, that is it looks at the contents of the address which are in the pointer. The following example is a loop which sets all 5 ports of the 16F877 to outputs and then sets all the port outputs to 0. We also haven't yet seen how loops are made up in C, in this example the `for` loop simply operates 5 times.

```

#include <P16F877.h>

void main()
{
    ADCON1=7;
    TRISA=TRISB=TRISC=TRISD=TRISE=0;

    char i;
    unsigned char *Port;

    Port=&PORTA;
    for(i=0; i<5; i++)
    {
        *Port=0;
        Port++;
    }

    endit:
    while(1);
}

```

8.1.11 sizeof

The `sizeof` operator returns the size of its argument in bytes. This is useful when allocating memory, or when designing programs which are to operate on a number of different processor types. For example:

```

char x,y;
long l;
char ram *rp;
char *p;

void main()
{
    y=sizeof(x);    // After this y is 1
    y=sizeof(l);    // After this y is 4
    y=sizeof(rp);   // After this y is 1
    y=sizeof(p);    // After this y is 2
    endit:
    while(1);
}

```

8.1.12 * / %

These operators perform multiplication (*), division (/) and modulus - the remainder (%).

```
x=y*z;
x=y/z;
x=y%16;
```

8.1.13 + -

These operators perform addition and subtraction:

```
x=y+z;
x=y-z;
```

8.1.14 << >>

These operators are also unusual to C. The expression on the left of the operator is shifted (bit by bit) left or right by the number of places on the right of the operator.

The left shift (<<) replaces the bottom bit(s) of the item by zeroes. The top bits which are shifted out of the value are lost. For example:

```
char y,x;
x=7;
y=x<<1;
```

In this case y is set to x with all of its bits shifted left by one place. Initially x is 7, which is binary 00000111, afterwards it will be 00001110 which is 14. The form x<<1 is equivalent to x*2 (x multiplied by 2).

The right shift (>>) copies the top bit of the item right for a signed value, and fills it with zeroes for an unsigned value. The bottom bits which are shifted out of the value are lost. For example:

```
char y,x;
x=0x31;
y=x>>2;
```

In this case y is set to x with all of its bits shifted right by two places. Initially x is 31 hex, which is binary 00110001, afterwards it will be 00001100 which is 0C hex.

The left shift can be used to set or clear bits in variables. The bitwise OR operator (|) performs a bitwise OR between the left item and the right item, and the bitwise AND operator (&) performs a bitwise AND between the left item and the right item. So if we wish to set bit 3 of port B we can use this form:

```
PORTB=PORTB|(1<<3);
```

Within the PIC there is an interrupt control register called INTCON. It has a bit called GIE which stands for General Interrupt Enable. To turn interrupts on GIE should be set, to turn it off it should be cleared.

Here is how to enable interrupts:

```
INTCON=INTCON|(1<<GIE);
```

... and here is how to disable interrupts:

```
INTCON=INTCON&~(1<<GIE);
```

Another way to write these is to use the assignment operators (see below):

```
INTCON|=1<<GIE;
```

```
INTCON&=~(1<<GIE);
```

Although this may look cumbersome WIZ-C Professional includes optimisations which make these forms very efficient in assembler code.

8.1.15 < <= > >=

These are the comparison operators (less than, less than or equal, greater than, and greater than or equal). They all return true (1) if the expression is true, and false (0) if it is not true.

```
char x=7,y=-1;
char r;

void main()
{
    r=x>=7;           // After this r is 1 - true;
    r=x>7;            // After this r is 0 - false;
    r=x>y;            // After this r is 1 - true;
    r=x<y;            // After this r is 0 - false;
    r=x<8;            // After this r is 1 - true;
endit:
    while(1);
}
```

8.1.16 == !=

These are logical operators. == is the equivalence operator which returns true (1) if the two items are identical and zero (false) if they are different. != does exactly the opposite.

```
char x=7,y=-1;
char r;

void main()
{
    r=x==7;           // After this r is 1 - true;
    r=x!=7;           // After this r is 0 - false;
    r=x!=y;           // After this r is 1 - true;
    r=x==y;           // After this r is 0 - false;
endit:
    while(1);
}
```

8.1.17 &

These operators are bit wise operators. & performs a bit by bit AND operation between its arguments, ^ performs a bit by bit XOR operation between its arguments, and | performs a bit by bit OR operation between its arguments.

```
char x=7; y=15;
char r;

void main()
{
    r=x&3;            // After this r is 3
    r=x|0xff;         // After this r is 0xff
    r=x|y;            // After this r is 15
    r=x&y;            // After this r is 7
endit:
    while(1);
}
```

8.1.18 && ||

These two operators are conditional AND and conditional OR operators. && returns true (1) if both of its arguments are true (non-zero), and false (0) otherwise. || returns true (1) if either of its arguments are true, and false if they are both false:

```
char x=7,y=15,z=0;
char r;

void main()
{
    r=x && 3;           // After this r is true (1)
    r=z || 0;           // After this r is false (0)
    r=x || y;           // After this r is true (1)
    r=x && z;           // After this r is false (0)

    endit:
    while(1);
}
```

&&,|| are interesting in that they don't evaluate the complete expression if it is not necessary. Consider:

```
(x>3) && (y>4)
```

In this case if x is 2, then the program will calculate (x>3) which is false, and then because the complete expression is false it will jump to the end of the expression without evaluating (y>4). Only if (x>3) is true will (y>4) be evaluated. Similarly:

```
(x>3) || (y>4)
```

If (x>3) is true then (y>4) is not evaluated because the complete expression is true, only if (x>3) is false will (y>4) be tested.

This is important because it can be used to make up expressions which avoid the need for testing. This will become clearer as we learn more of the language, however here is an example (it uses keywords we haven't seen yet, but the sense should be clear, and you can revisit this example after the chapter on pointers to see why it is important):

```
if (xp!=0)
    if (*xp==1) y=2;
```

This can be rewritten as:

```
if (xp && *xp==1) y=2;
```

8.1.19 ?:

C has a conditional operator which has three components. It is written as:

```
Test ? Expression if Test True : Expression if Test False;
```

If the test is true (non-zero) then the first part of the expression (before the colon) is returned, if the test is false (zero), then the second part is returned.

For example :

```
char x=1,y;

void main()
{
    y=(x>0) ? x : -1;           // After this y will be x (1)
    y=(x==0) ? x : -1;         // After this y will be -1

    endit:
    while(1);
}
```

```
}
```

8.1.20 =

The = operator simply assigns the value on the right of the equals sign to lvalue on the left of the item. We have seen plenty of examples of the = assignment so far so no new examples are shown here.

It is notable that the equals expression returns the value on the right hand side which can be used to set several values at once. For example here is one way to reset all the ports of the PIC 16F877 to the value 0:

```
PORTA=PORTB=PORTC=PORTD=PORTE=0;
```

8.1.21 *= /= %= += -= &= ^= |= <<= >>=

These are the assignment operators. The best way to explain them is through an example:

```
x+=7;
```

This is equivalent to:

```
x=x+7.
```

In other words the lvalue on the left is assigned to a new value which is calculated from the value on the left operated with the value on the right.

```
x*=y;      // is equivalent to x=x*y
x/=y;      // is equivalent to x=x/y
x%=y;      // is equivalent to x=x%y
x+=y;      // is equivalent to x=x+y
x-=y;      // is equivalent to x=x-y
x&=y;      // is equivalent to x=x&y
x^=y;      // is equivalent to x=x^y
x|=y;      // is equivalent to x=x|y
x<<=y;     // is equivalent to x=x<<y
x>>=y;     // is equivalent to x=x>>y
```

As for the = operator the assignment operators return the value on the right hand side of the expression.

8.1.22 ,

The comma operator is not often used. The comma will separate a number of expressions, the value returned is the right most value.

```
y=(x>2),3;      // y is set to the value 3
```

8.2 Operator precedence

The operators work in strict precedence order. Where two operators have the same precedence it is not possible to predict the order of evaluation. In all cases precedence of operators can be forced by using brackets ().

The precedence order from highest to lowest is as follows:

1	() [] -> .
2	! ~ + - ++ -- & * sizeof
3	.* ->*
4	* / %
5	+ -
6	<< >>
7	< <= > >=
8	== !=
9	&
10	^
11	
12	&&
13	
14	?:
15	= *= /= %= += -= &= ^= = <<= >>=
16	,

9 Functions

9.1 Introduction

So far we have only looked at very simple programs. However every program we have looked at includes a single function called main. A function is a block of code which can be called (or run) from other code blocks. Each function (code block) stands alone - you cannot force a program to jump into the middle of a function, although functions can return (jump back out) from any point within the complete function.

Use the project Functions.PC which can be found in the Projects\Learn To Use C\09 - Functions directory within the WIZ-C Professional installation.

9.1.1 Functions Described

9.1.1.1 Declaring functions.

Functions can return values which may be used in expressions. Functions must be declared before they are called from other functions - this is similar to declaring variables.

Here is an example of how to declare functions which return values:

```
char GetValue();           // Function returns a character
unsigned int ReadAToD();   // Function returns an unsigned int
```

Note that the compiler knows that we have a function because there are brackets () after the name. Now we could use these functions in expressions:

```
unsigned int Voltage;
char Value;

Voltage=ReadAToD()*2;
Value=GetValue();
```

You can also simply call the function without using an expression:

```
GetValue();
ReadAToD();
```

Functions may do not have to return a value. A function which returns no value is called a void function, main is an example of a void function. Here are some examples of how to declare void functions:

```
void TurnOnLED();
void TurnOffLED();
```

void functions may not be called in an expression:

```
TurnOnLED();           // This is valid
x=TurnOnLED():         // This will give a syntax error
```

Now we can look at how to write a function, the type of function followed by its name (and the brackets) starts the function. Then all the code for the function is included within curly brackets. We have seen this in all the example programs for main(). Here is the function which turns on an LED connected to pin RB0 (PORTB, bit 0).

```
void TurnOnLED()
{
    TRISB&=~1;           // Clear bit 0 of TRISB to drive pin RB0
    PORTB|=1;            // Turn on LED
}
```

Note that as this is a void function there is no return value. To return a value from a function then the return keyword is used followed by the value to be returned. Here is the GetValue function which returns the value of PORTB.

```
char GetValue()
{
    TRISB=0xff;    // All pins of port B are set to read
    return PORTB;  // return the value of port B
}
```

A void function may contain a return (with no value), however functions which return values must include a return statement with a value to be returned at the end.

Note that a function can contain as many return statements as required - this allows functions to return early when conditions are not met or errors are encountered.

9.1.1.2 Parameters

Functions can be passed parameters - values which are supplied when the function is called, and which can be used as variables within the function. The parameters of a function have no meaning outside that function.

Function parameters are declared within the brackets of the function definition. They may have any type and any legal variable definitions are legal within a function definition. Here are some function definitions with parameters:

```
void Delay(int x);           // Delay for x milli-seconds
char SerialTx(int x);        // Transmit a value x, return acknowledgement
int Square(int x);           // Square the value x - return squared value
void Clear(char x[],int n);   // Clear the first n values of array x
```

Now here is how these functions could be called:

```
Delay(1000);                 // Delay for one second
Ack=SerialTx(7);             // Transmit value 7, set Ack to acknowledgement
S=Square(Ack);               // Set S to Ack*Ack
Clear(Array,7);              // Set first 7 values of Array to 0
```

Defining the function is similar to functions without parameters. Here is the function Square:

```
int Square(int x)
{
    return x*x;
}
```

9.1.2 Examples of functions

Let's have a look at some examples of functions. Firstly in this program the function Mult2 simply multiplies the value it is supplied by 2 and returns the new value.

```
char x=1,y=2;

char Mult2(char v);          // Declare the function

void main()
{
    x=Mult2(x);
    y=Mult2(y);
endit:
    while(1);
}

char Mult2(char v)
{
    v=v*2;
    return v;
}
```

After this program completes x will be 2 and y will be 4.

If you try single stepping this program and watching the variables x, y and v you will see that v is not recognised until you single step into Mult2 at which point it will spring into life and show the value 1 and then 2 as it is called by x and y.

Now let's have a look at something a bit more useful. This program sends an 8 bit clocked value on to port B on any bit. The function ClockOut takes two parameters - the value to be clocked, and the bit number from 1 to 8, the clock will be sent on bit number 0. It is a void function - that is it returns no value, and therefore requires no return statement (although a return statement can be used with no return value).

For our example we used a 16F88.

Again we will use a for loop - we'll look at these in a later chapter.

```
#include <pic.h>

void main();
void ClockOut(unsigned char val, unsigned char Bit);

void main()
{
    ANSEL=0;          // This sets all pins to digital on the 16F88
    TRISB=0;          // Set PORTB to all outputs
    PORTB=0;          // Clear PORTB
    ClockOut(0x55,1); // Send value 0x55 to PORT B, bit 1

endit:
    while(1);
}

//
// Send a serial value out to a port
//

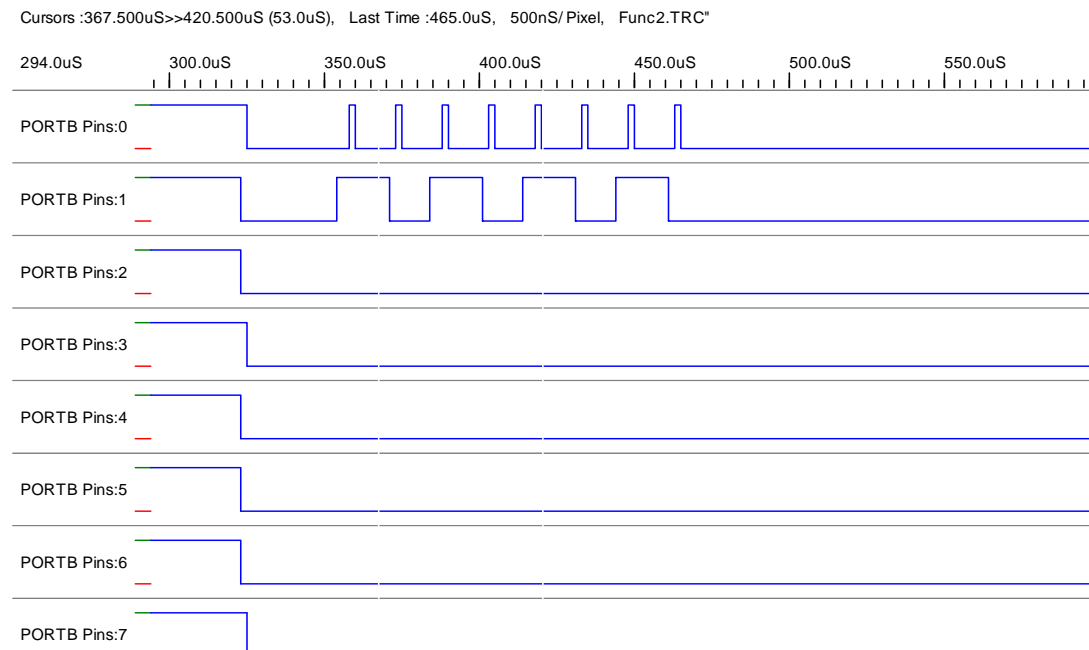
unsigned char i;          // used as a loop counter
unsigned char dmask;      // Mask for the bit used to transmit data
const unsigned char dclock=1; // Mask for the clock bit which is bit 0

void ClockOut(unsigned char val,unsigned char Bit)
{
    dmask=1<<Bit;        // This is the mask for the data bit

    for(i=8; i; i--) // Loop 8 times
    {
        if (val&1) PORTB|=dmask; else PORTB&=~dmask; // Set or clear data bit
        PORTB|=dclock; // Send clock high
        PORTB&=~dclock; // Send clock low
        val>>=1; // Shift value right one place
    }
}
```

Use a new project (call it SerialOut – we have provided this in the 09 – Functions directory.) and enter this code into the main program. Compile it for a 16F88 and set a breakpoint at the endit label. Set the simulation update rate to a high value. Single step using F7 to see how the pins change from Analogue input (Blue) to digital input and are then set to outputs, press F9 to run to the breakpoint.

Start the waveform analyser. Add Port B as 8 line traces and use F8 to zoom in and check that you can see the waveform at around 300uS (the earlier gap is whilst the program clears RAM memory).



9.1.3 Recursive functions

Recursive functions are those which call themselves - either directly or through another function.

A classic example of a recursive function is working out the factorial of a number (this is $x*(x-1)*(x-2)\dots 1$). Factorial of 2 is 2 times 1. Factorial 3 is 3 times 2 times 1. We can write a recursive function to calculate the factorial of a number. If the supplied number is 1 then we return 1, if the supplied number is x (when x is not 1) then we return $x*\text{Factorial}(x-1)$.

Look at the example - WARNING please don't attempt to compile and run this program until you have read the instructions below it. Create a new directory and project for this - call it Factorial (we called it Factorial in the 09 – Functions directory).

```
int fact(int v); // Declare factorial function
int x;

void main()
{
    x=fact(1);
    x=fact(2);
    x=fact(3);
    x=fact(4);
    x=fact(5);
    x=fact(6);
    x=fact(7);
    endit:
    while(1);
}

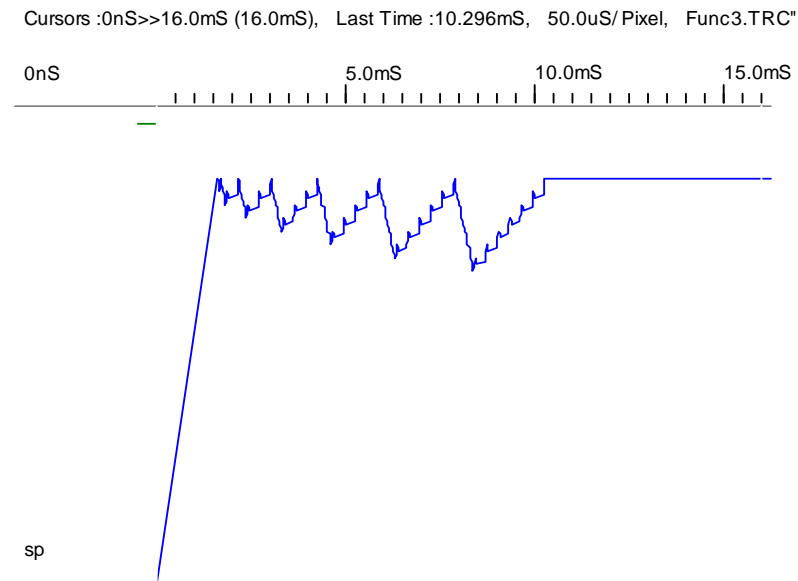
int fact(int v)
{
    if (v==1) return 1; // Factorial 1 is 1
    return v*fact(v-1); // Otherwise calculate v*factorial(v-1)
}
```

When compiling this program first select a 16F877. Secondly on the Compiler Options dialog box select the Optimisations tab and clear the tick by "Used PIC Stack (Quick Call)". This is essential because the fact function calls itself, and each time it calls itself it uses the stack. All our programs so far have used the PIC stack which is only 8 items deep. Our function fact

would overflow the internal stack and cause a crash which is why we disable use of the PIC Stack and instead use the software stack (which is regrettably a bit slower).

Run the program and check that x is 5040 which is factorial 7. (You can also step through the main function by resetting it and pressing F8 to execute each call one line at a time).

Click the sp variable and select the trace box. Reset the processor and run again. Now select the waveform analyser and view sp as an analogue trace - drag the line below sp down the screen to make the trace bigger. Can you see the software stack (which is held in sp) moving down for each factorial call ?



9.2 Variables in functions

9.2.1 Local variables

9.2.1.1 What are local variables ?

Functions can have their own variables. Any variable declared within a function is called a local variable, and can only be used within that function. If a variable is declared within a function and outside it, then if the variable is used within the function then it is the local variable which will be used.

Here is the ClockOut function again, but this time using local variables:

```
//
// Send a serial value out to a port
//

const unsigned char dclock=1;    // Mask for the clock bit which is bit 0

void ClockOut(unsigned char val,unsigned char Bit)
{
    unsigned char i;              // used as a loop counter
    unsigned char dmask;          // Mask for the bit used to transmit data
    dmask=1<<Bit;                // This is the mask for the data bit

    for(i=8; i; i--) // Loop 8 times
    {
        if (val&1) PORTB|=dmask; else PORTB&=~dmask; // Set or clear data bit
    }
}
```

```

        PORTB|=dclock;           // Send clock high
        PORTB&=~dclock;         // Send clock low
        val>>=1;                 // Shift value right one place
    }
}

```

Using local variables within functions is good programming practice. Local variables don't use up valuable space when the function is not running. Local variables allow functions to call themselves (to be recursive), each instance of the function with its own copy of the variables. Finally local variables allow names to be reused without worrying about whether they are already in use.

9.2.1.2 Debugging with local variables.

There are some downsides to the use of local variables. They are slower to use than normal variables (unless they have been optimised to global memory - we'll look at this), they are also much harder to debug – unless you have the professional version of the compiler.

Enter the following test program (you can use the vars project, or create a new project, this is saved in the FED examples Func4).

```

char x=1;

char Mult2(char v);      // Declare the function

void main()
{
    x=Mult2(x);

endit:
    while(1);
}

char Mult2(char v)
{
    char w;
    w=v;
    w=w*2;
    return w;
}

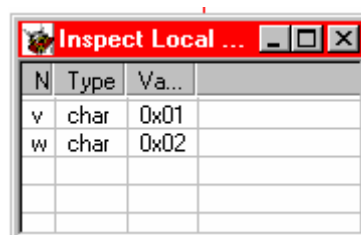
```

This program multiplies x by 2 in a very roundabout way.

Now when compiling this program (use the 16F84) click on the Optimisations tab and in the box titled "Local Optimise Bytes", enter the value 0. (this turns off one of the optimisations as we'll see later). Make sure the "Use PIC Call Stack" option is selected (ticked) as well.

9.2.1.2.1 Professional Version.

If you have the professional version you can watch the stack variables in C format directly. If you can't see the Inspect Local Variables dialog box then right click the debugging window (Watch tab) and enable it using the menu option. Single step the program using F7 and watch the variables change :



You can ignore the sections below unless you wish to see how the stack and local optimisation works.

9.2.1.2.2 Local variables on the stack – normal version.

Now press F7 and the program will run to the first line in Mult2. Local variables and parameters are stored on the software stack. The variable sp is the stack pointer for the software stack - it points to the first free byte on the stack. It will show 4E hex. On the 16F84 the software stack starts at 4F hex and works downwards (the C Compiler manual shows all this in more detail).

Examine 16 bytes of memory from address 0x40 which will show the last bytes in RAM leading up to the stack. You may already have a watch in the debugging window which shows 16 bytes from 0x40. If not click the debugging window, Watch tab. Press Insert and enter the value 0x40. Make sure Hex Dump and Hex are displayed.

So as sp is 4E hex then we know that there is one item on the stack (recall sp starts at 4F). Look at the memory from 0x40 - the last byte displayed shows 1 - this is the value of v within the function Mult2, it was placed there by the call `x=Mult2(x);`

Now press F7 again - note that sp will be reduced by 1 to show 4D. This is because the compiler is making space on the stack to store the variable w.

Press F7 again and this time v will be copied to w - there will be two consecutive bytes both of value 1 at the top of the stack.

Press F7 and w will be multiplied by 2 again watch the stack.

Press F7 again and x will be set to 2, and the program will return to the main function at its last line. Check sp - it will be back to 4F hex.

9.2.1.2.3 Optimised Local variables

Stack variables are a little slow - particularly on the PIC which has no hardware stack for data. WIZ-C Professional optimises local variables to a memory space called LocOpt wherever possible. The details of how it does this are covered in the main C Compiler manual.

Compile this program (use the 16F84), however this time click on the Optimisations tab and in the box titled "Local Optimise Bytes", enter the value 4. This will allocate 4 bytes for local variables (the bigger this number the better, but it does use valuable memory space).

Now open the file Func4.asm (or whatever your main file is called), scroll down until you find the label for Mult2, it should look something like this:

```

;~ Line : 00014 char Mult2(char v)
00014
00014
00015 ;~ Line : 00015 {
00015
00015
00015 ; Function : Depth=0,LocOpt Size=2,Locals Offset=0
00015 ; Calls:
00015 ; Parameter v optimised to (LocOpt+1)
00015 ; Local Variable w optimised to (LocOpt+0)
00015 module "Mult2"
00015 Mult2::
00015
00016 ;~ Line : 00016 char w;
00016
00016
00017 ;~ Line : 00017 w=v;
00017
00017 movfw (LocOpt+1)

```

```
00017  
00017 movwf (LocOpt+0)
```

The lines above the label Mult2 are optimisation information for the function. In this case they are telling us that parameter v can be found at address LocOpt+1, and the local variable w can be found at address LocOpt+0. Look at the watch called LocOpt (if you haven't got one then add a watch at address LocOpt and set to hex dump). Note that the first byte is 0, and the second byte is 1 - this is the parameter v.

Single step using F7 and watch how w and v change in the LocOpt line.

9.3 Exercises

1. Modify the ClockOut example so that the function can output the clock and the data on any pin of PORTB.
2. Run the factorial example without clearing the "Use PIC Stack (Quick Call)" option. Single step it. How far does the program get before crashing ?
3. Write a simple example program using a function with 2 parameters and no local optimisation space. Debug it and watch the stack - which variable is stored where ?

10 Program control

So far we have looked at variables and functions. In this section we'll look at program control - those keywords which affect the flow of a program, but strictly within a function. The keywords for flow control can only be used within a function, and can not transfer control out of a function - the function is a self contained module which can only be entered from the top by a normal function call.

Use the project Control.PC which can be found in the Projects\Learn To Use C\10 - Control directory within the WIZ-C Professional installation.

10.1 if and else

The if and else statements are similar to those in other languages, the if test is very simple:

```
if (condition) statement;
```

If the condition is true then the statement is executed, otherwise control jumps to the next statement. There is an optional else statement:

```
if (condition)
    statement1;
else
    statement2;
```

If the condition is true then statement1 is executed after which control jumps to the statement after statement2, otherwise control jumps to statement2.

Here is an example function which calculates the absolute value of an integer:

```
int abs(int v)
{
    if (v<0) v=-v;
    return v;
}
```

Here is a function which returns the ASCII character for a hex value - i.e. if the value is from 1 then the ASCII value of 1 (which is 49) is returned. The input value is from 0 to 15, if it is outside this range then the character 'X' is returned.

```
char HexChar(unsigned char v)
{
    if (v>15) return 'X';
    if (v>=0 && v<=9) v=v+'0';
    else v=v-10+'A';
    return v;
}
```

Note that this is not the most efficient version possible, however it does demonstrate use of if and else.

10.2 for

Within C the for statement is similar to that used in other languages. There are three sub-statements in a for statement:

```
for(Part1;Condition;Part3) Action_Statement;
```

Part1 is executed first, and once only before the loop begins. Condition is a test - if the test is true the loop runs, if it is false then the program jumps beyond the end of the loop. Part3 is

the action to be undertaken at the end of each loop. Finally the Action_Statement is a single statement (or compound statement) which is executed on each loop.

To compare with the BASIC language here is a For loop in BASIC:

```
for i=0 to 10 step 1
  Code
next i
```

Here is the equivalent in C

```
for(i=0; i<=10; i++)
{
  Code
}
```

Identify parts 1, the condition, and part 3 for the statement and compare with the description above to see how the BASIC and C versions are identical in their operation.

Here is an example which adds all the elements in an array called a and stores them in a variable called Sum:

```
char a[10];          // Array
char Sum;            // Sum of elements

Sum=0;
for(i=0; i<10; i++) Sum+=a[i];
```

Here is an example which does the same calculation, but also sets all the elements of the array back to 0:

```
char a[10];          // Array
char Sum;            // Sum of elements

Sum=0;
for(i=0; i<10; i++)
{
  Sum+=a[i];
  a[i]=0;
}
```

Here is an example which on a two dimensional array sets all the elements to 0xff. This example uses two for loops:

```
char a[3][4];
unsigned char i,j;

void main()
{
  for(i=0; i<3; i++)
    for(j=0; j<4; j++)
      a[i][j]=0xff;

  endit:
  while(1);
}
```

Try this program in the C Compiler and simulator - note how the elements of the 12 byte array are set.

The statement at the end of the loop (Part3), can be any c expression or statement. Here is an example which sets the 8 elements of an array (a) to their bit values (1,2,4,8 etc). See if you can see how it works.

```
char a[8];
unsigned char i,j;

void main()
{
```

```

j=0;
for(i=1; i; i<=1) a[j++]=i;

endit:
while(1);
}

```

10.3 while

A while loop executes the statement whilst the condition is true:

```
while(condition) Action_Statement;
```

For example this program will calculate the length of a string (remember a string is a character array, and the last value at the end of the string is a 0 byte).

```

char s[16];
unsigned char i,len;

i=0;           // Looks along the string
len=0;         // Length of string
while(s[i])    // Loop until an array variable is zero
{
    len++;      // Add one to the length
    i++;        // Add one to the index (look at next element)
}

```

If the statement is not true then the loop does not execute - it is possible for the loop never to run if the condition is false when the while statement is tested for the first time. Here is an example program to add together all the numbers from 1 to 10:

```

unsigned i,s;

void main()
{
    s=i=0;

    while(i<11) {s+=i; i++;}

endit:
while(1);
}

```

10.4 do

The do loop is similar to a while loop, except that the condition is tested at the loop end, so a do loop always executes at least once. do loops are used less often than while and for loops.

```

do
    Action_Statement
while(condition);

```

Note that the while statement at the end of the loop must end with a semi-colon. Here is the program to add all numbers from 0 to 10 again, but using a do loop:

```

unsigned char i,s;

void main()
{
    s=i=0;

    do
    {
        s+=i;
        i++;
    }
    while(i<11);
}

```

```

endit:
while(1);
}

```

10.5 goto and labels

It is possible to transfer control using a goto statement which unconditionally jumps to a label within the same function.

A label is any alphanumeric string ending in a colon. Here is an example which sets all values of an array to 0xff.

```

char a[10];

void main()
{
    char i=0;

loop:
    a[i++]=0xff;
    if (i!=10) goto loop;

endit:
    while(1);
}

```

Note that in all our examples we have used the label endit. This has always been to enable a breakpoint to be set - so far we have not directly jumped to the end label.

10.6 continue and break

For all the loop statements that we've looked at, it is possible to break out of the loop using the break keyword. This simply takes the next loop up (be it for, do or while), and jumps to the statement after the loop. Here is a code fragment example which waits for bit 0 of PORTC to be at level 1 before breaking from the loop:

```

while(1)
{
    Count++;
    if (PORTC&1) break;
};

```

The continue keyword jumps within the loop (again do, while or for). If the loop is a for loop then the continue jumps to the 3rd part of the for statement and then back to the conditional. If the loop is a while loop then it jumps back to the conditional to test whether to run the loop again, or jump to the end. Finally for a do loop the continue keyword jumps back to the top of the loop. Here is a code fragment example:

```

while(Flag)
{
    if (PORTC&1) continue;
    PORTB=PORTB^1;
}

```

10.7 switch and case

The final program control that we'll look at is the switch statement. A switch statement takes a value and compares it to an number of case values. When they match the code by the case is executed:

```

switch(ValueIn)
{
    case test1: Code1.... break;
    case test2: Code2.... break;
    default : Coded...; break;
}

```

```
}
```

If ValueIn matches the value test1 then Code1 is executed. If it matches test2 then Code2 is executed etc. If it matches none of the test values then the default code is executed. In all cases if there is no break then the code runs on into the next case statement.

Here is an example program. This program operates on the 16F877 development board. It uses the library routine SerialIn() - this is a function which reads a value from a port pin which is transmitted in serial format compatible with a PC. In this example if an 'A' character is read from the port then the first LED on the board is illuminated. If 'B' is received the second LED is illuminated, 'C' will light the third, and 'D' the fourth. Any other character received will turn off all the LED's. The received character is transmitted back to the terminal.

```
#include <pic.h>
#include <datelib.h>

#_config _CP_OFF & _BODEN_ON & _WDT_OFF & _PWRTE_ON & _HS_OSC & _LVP_OFF

const long SERIALRATE_IN=9600;           // Serial rate
const int BITTIME_IN=APROCFREQ/SERIALRATE_IN/4; // Time for 1 bit
const int BITTIME_OUT=APROCFREQ/SERIALRATE_IN/4; // Time for 1 bit
const BYTE SERIALPORT_IN=&PORTC;         // Port for serial input
const BYTE SERIALBIT_IN=7;               // Bit for serial input
const BYTE SERIALPORT_OUT=&PORTC;        // Port for serial output
const BYTE SERIALBIT_OUT=6;              // Bit for serial output

void main()
{
    char rx;

    ADCON1=7;           // Set ports A and E to all digital I/O's
    TRISE=3;            // Drive transistor for LED's
    PORTE=4;            // Turn on transistor for LED's
    PORTD=0;            // Set all LED's off
    TRISD=0x0f;         // Drive LED outputs (bits 4-7 of PORTD)
    TRISC&=~(1<<SERIALBIT_OUT);

    while(1)
    {
        rx=pSerialIn();
        pSerialOut(rx);
        switch(rx)
        {
            case 'A' : PORTD|=0x10; break; // Turn on LED 1
            case 'B' : PORTD|=0x20; break; // Turn on LED 2
            case 'C' : PORTD|=0x40; break; // Turn on LED 3
            case 'D' : PORTD|=0x80; break; // Turn on LED 4
            default  : PORTD=0; break;     // Turn off all LED's
        }
    }
}
```

Create a new project in a new directory for this program – it is supplied with the FED examples as Control. Compile it for the F877 running at 20MHz.

10.7.1 Simulate using external devices

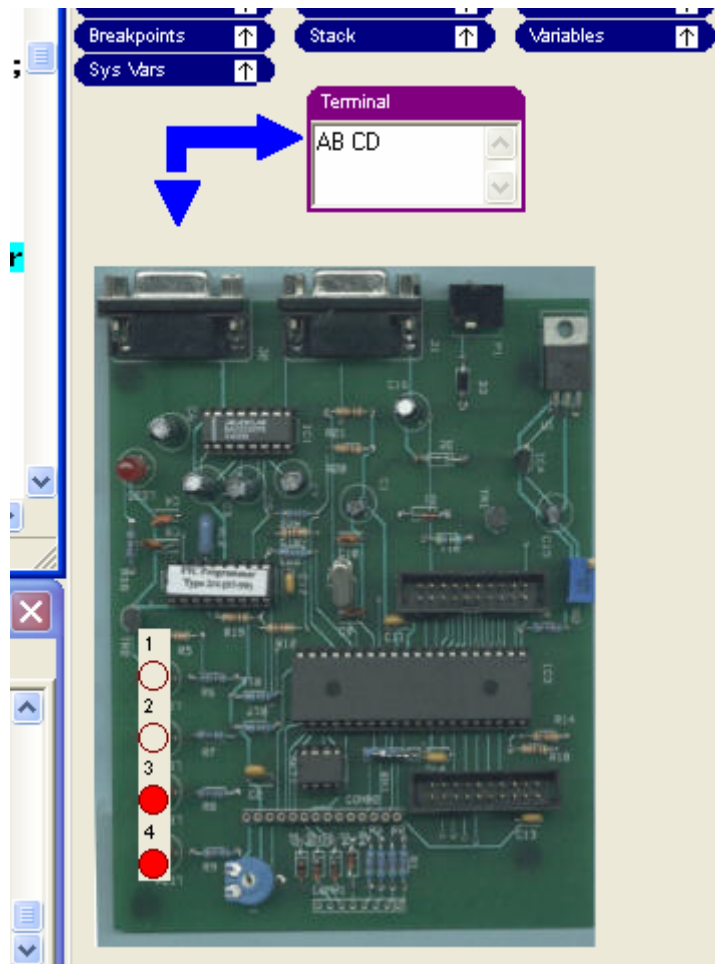
Add four LED's to Port D pins 4 to 7 using the **Simulate | Add external device** menu option (the technique to do this is shown in chapter 4, however we can ignore the drive from Port E2 – simply connect the cathodes low).

Now we can add a terminal device.

Use the **Simulate | Add external device** menu option and select the Terminal type. The Terminal has two connections. TerminalRx is the pin on which the terminal receives data, it needs to connect to the PIC transmit pin on PORTC bit 6. Similarly TerminalTx connects to

the PIC receive pin on PORTC bit 7. Connect the terminal to these two pins. The terminal needs to be set up to run at 9600bps. Look in the parameters box, select the Bit Rate option, and from the drop down box select 9600bps.

You can drag the devices around so that you can see them in any way you wish - we also took advantage of the ability to add bit maps to the debugging window to show the LED's on the correct position on the board. There is a bit map of the FED development board in the Images sub-directory of the PIXIE installation.

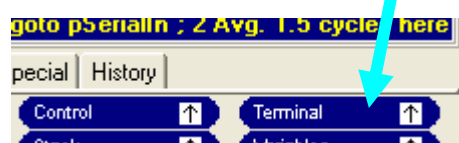


The LED's may all be on at program reset – don't worry about this, they will go off as the program runs).

Run the program using a high update rate – type characters A,B,C,D and E into the terminal – watch the LED's come on and then all go out.

10.7.2 Run on the development board

If you have the development board you can try the program directly - the PC serial lead will need to be plugged into J2 (the lower connector) on the board. The C compiler includes a terminal emulator which can be used for this program. Click on the debugging window and then on the Terminal Window at the top right of the debugging window :



This is a real terminal which may be connected to any of the PIC's communication ports.

Right click the window and use the **Setup Terminal** menu option. Set the terminal to 9600bps and use the communications port to which the serial lead and development board are connected. Press shift and A (remember that capital letters are used here) and check the correct LED illuminates - repeat with other keys.

Once you have finished remember to disconnect the serial port by right clicking the terminal window and using the **Disconnect** menu option.

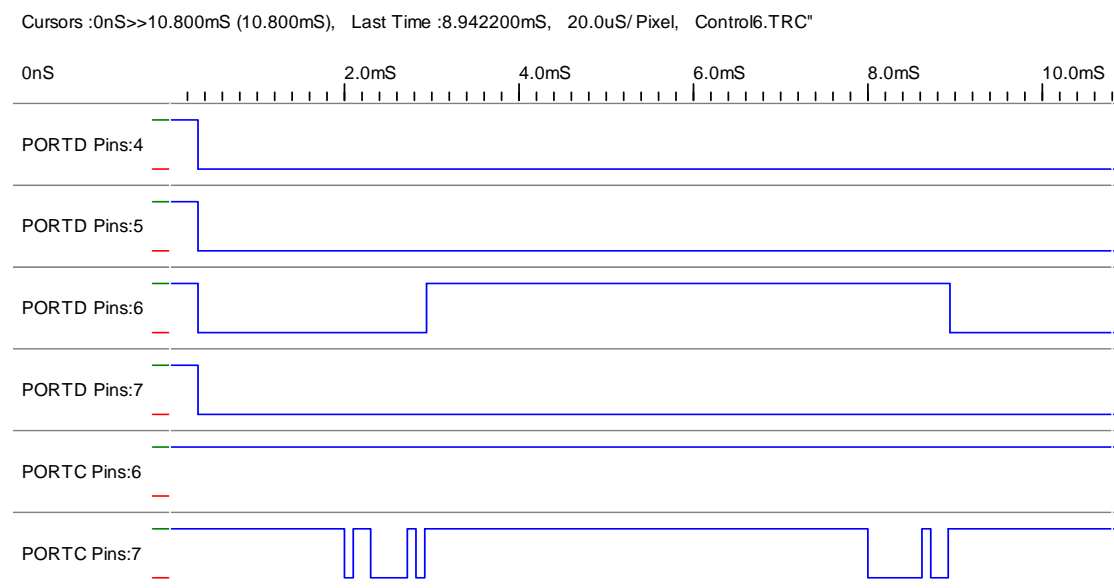
10.7.3 Simulate using stimulus files

You can also simulate this program using stimulus files. Open a new file and save it as "Input.STI" (the save as dialog box has the Save As drop down box, select "Stimulus" files from this box). On the project window click the STI/Inject files tab. Press insert. Select Files of Type and Stimulus. Select the Input.STI file. Now into the file "Input.STI" enter the following and then save the file:

```
2m
serial9600-PORTC:7=43H          ; Character 'C'
8m
serial9600-PORTC:7=20H          ; Space character - turn off all LED's
```

This will inject a serial data string into port C, bit 7. The first character is a 'C', the second is a space to turn off the LED's. The characters will be sent at 2mS and 8mS after the simulation begins. Run the program to 10mS - you should see PORTD change from 0xff, to 0x0f, to 0x40 and finally back to 0x0f. (This represents LED 3 turning on and off - LED 3 is on Port D, bit 6).

This can be examined on the waveform analyser window - this will display a graphical view of the output. Use the **Tools | Examine Wave Window** menu option. A dialog box will appear "Define Trace Format". Click the trace name box and select "PORTD Pins", click the "Add as 8 Line Traces" button. This will show all pins of Port D, but the time scale will be so short that nothing will appear to be happening. Press F8 to zoom in - and then press it again and again and again until you can see Port D bit 6 turn on and off again. Here are the relevant signals – watch the incoming bytes and Port D bit 6



11 Pointers

11.1 Introduction

The concept of pointers is probably the hardest to pick up in C, even so it is not too difficult to understand the concepts.

Use the project Pointers.PC which can be found in the Projects\Learn To Use C\11 - Control directory within the WIZ-C Professional installation.

11.2 Definition

A pointer to anything does not hold a value, but holds the address of a variable or value. If we have a character variable x, then a pointer to x holds the address of x. We'll start by looking at how to declare and use pointers.

Firstly to declare a pointer then use a type with a star (*) in front of the name. To declare a pointer to a character value then this code can be used:

```
char *cp;
```

As before many items can be declared including pointers on the same line. Here is how to declare a pointer to a character and a character variable:

```
char *cp,x;
```

Now the pointer cp does not point at anything - if it is a global variable it has the value 0, and so points to location 0 in memory - not terribly useful. The address of any lvalue can be read by using an & symbol in front of the expression. Here is how we set cp to point to x:

```
cp=&x;
```

Finally in this simple review of pointers, it is possible to set the contents of the variable to which a pointer is pointing using the * operator. Here is how to set the contents of memory to which cp is pointing to the value 7:

```
*cp=7;
```

Here is a simple example of the use of pointers. There are two variables x and y, and a pointer cp. cp is set to point to x, its contents are set to 7, and y is then also set to the contents of the value pointed to by cp.

```
char x,y,*cp;

void main()
{
    cp=&x;
    *cp=7;
    y=*cp;

    endit:
    while(1);
}
```

Try this program out. Watch the value of x, y and cp. cp will probably be set to 34 hex on the F877(or somewhere close), this is the address of x. Here are the values of cp, x and y after it has run, note that we are using the professional version and displaying using C format:

C cp	M 34 ->0x07
C x	M 0x07
C y	M 0x07

This was not terribly useful. However one good use of pointers is to change the values of variables within functions. Here is an example which uses a function called PortSet to set all bits of a port to value 0, then all bits to value 1 and then back to 0 again. This example is for the 16F84.

```
#include <P16F84.h>          // Include 16F84 header specifically

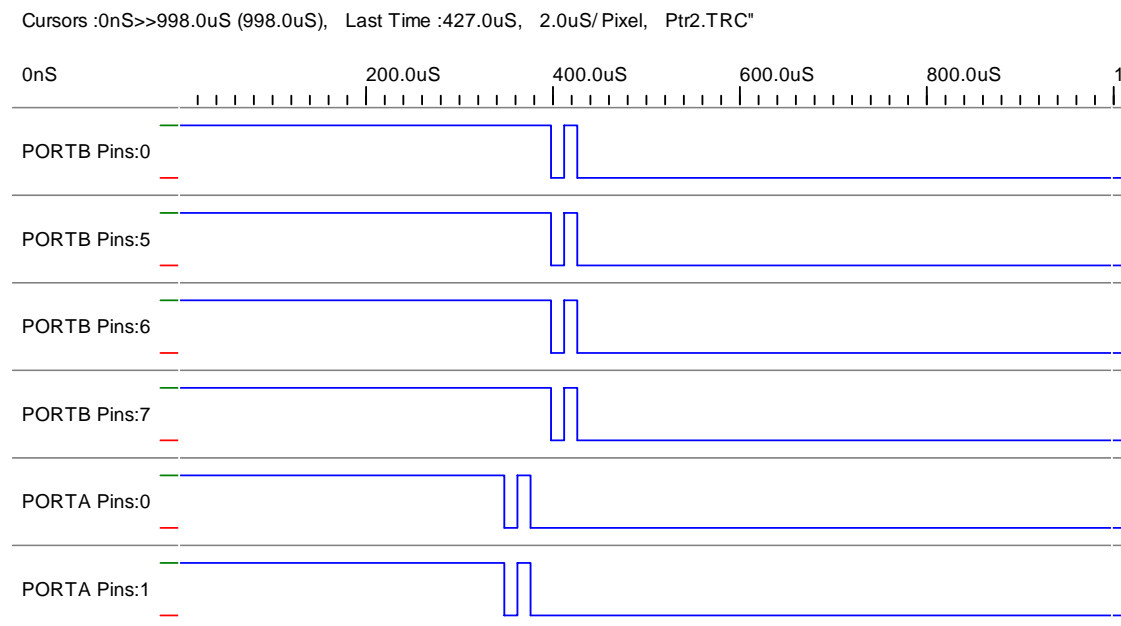
void PortSet(unsigned char *Port);

void main()
{
    PORTA=PORTB=0xff; // Start with all bits set to 1
    TRISA=TRISB=0;

    PortSet(&PORTA);
    PortSet(&PORTB);
endit:
    while(1);
}

void PortSet(unsigned char *Port)
{
    *Port=0;           // Set pins of port to 0
    *Port=0xff;        // Set pins of port to 1
    *Port=0;           // Set pins of port back to 0
}
```

Create a new project to test this program (Ptr 2 on the FED examples). Use the waveform analyser and check that PORTA and then PORTB perform as expected. Here are some pins from Port A and B.



11.3 Arrays

A pointer to the first element of an array can be used as if it were the array and referenced in the same way as the array. Similarly an array variable when taken without the array index can behave in the same way as a pointer. For example:

```
char a[10];
char *ap;

ap=a;      // Now ap and a point to the same array
ap[0]=7;   // Set first variable of array a to value 7
ap++;
```

```
ap[0]=8;    // Set second variable of array a to value 8
```

Note that a pointer is more capable than the array variable - the following is illegal:

```
char a[10];
a++;
```

The pointer can be incremented to point to the second variable in the array, but the array variable cannot be incremented.

11.4 Pointers and strings

One big use of pointers is within strings. Recall that a string in C is an array of characters terminated by a 0 character. Therefore a pointer to a character can be used to perform operations on the string array. Here is a function which will copy a string from one location to another:

```
char *StringCopy(char *t,char *f)
{
    char *result=a;

    while(*f) *t++=*f++;
    *t=0;
    return result;
}
```

This function copies a string from one pointer to another. Note that whilst the pointer f contains a non-zero character then it will be copied to the pointer t. When f contains the zero character (at the end of the string) then copying is ended and a zero byte is copied to location t to end that string. Finally the returned value is the original value of t - a pointer to the copy of the string. Here is the full test program:

```
char *StringCopy(char *t,char *f);

char a[10];

void main()
{
    StringCopy(a,"Test");
endit:
    while(1);
}

char *StringCopy(char *t,char *f)
{
    char *result=a;

    while(*f) *t++=*f++;
    *t=0;
    return result;
}
```

If you would like to test this program (saved as Ptr3 in the FED example files) then examine the string (character array) "a" as a hex dump and as a string. Note that the array a can be passed to the StringCopy function as a pointer - within the StringCopy function the copy of a (called t) can be used as a pointer and incremented. Secondly the string "Test" can be passed to the function - it is held within PIC ROM memory and copied from there.

The ability to copy one string to another is very useful and is provided in a library function. The function (strcpy) is included in the string library. To tell the system about the functions in the string library we need to include the library header - this is the first line of the following example program which does exactly the same as the previous example:

```
#include <string.h>

char a[10];
```

```

void main()
{
    strcpy(a,"Test");
endit:
    while(1);
}

```

If you'd like to see all the string functions then click the cursor on the strcpy word and press Control and F1 together.

It is possible to have pointers to pointers, pointers to arrays, and arrays of pointers. Here is an example of a program which is incomplete, it creates an array called Options which is an array of pointers to strings. The function "display" is called to display a string on a screen. In this version the program only calls the display function once with the first string, and the function display simply copies the string to the array "a" so that it can be viewed in the debugger (Ptr4 in examples):

```

#include <strings.h>

const char *Options[]= // Array of option strings
{
    "Start String",
    "Option 1",
    "Option 2",
    "etc."
};

void display(char *String);
char a[16]; // Array to "display" selected string

void main()
{
    display(Options[0]); // Display first option

endit:
    while(1);
}

//
// Function to display a string somewhere - in this case we simply copy it
// to the array "a"
//

void display(char *String)
{
    strcpy(a,String);
}

```

Note how the array of pointers is initialised - in this case the empty square brackets, [], simply tell the compiler to declare an array big enough to hold all the items in the initialisation, this is 4 items in this example. Note also that Options is a constant array – this forces the pointers and the strings into ROM and therefore the Options array and its strings take no RAM space.

11.5 Pointer Arithmetic

Operations can be performed on Pointers in the same way as other variables. However the compiler knows to that type of object a pointer is pointing, and undertakes arithmetic slightly differently. For example:

```

int a[5];
int *ap;

void main()
{
    ap=a;
    *ap=0;
}

```

```

    ap++;
    *ap=1;
endit:
    while(1);
}

```

In this example ap is set to point to the first item of array a, and this item is set to 0, then ap is incremented and therefore the second item of array a is set to 0. However an integer is 2 bytes long, so the actual value of ap is increased by 2. The compiler handles this for us completely automatically and values can be added and subtracted from pointers and the pointer will still be correct. Run this program and watch ap as it is incremented to confirm this. You may like to extend the program to add different values to ap. (Use the += operator). If you have the professional version you can display a as a C variable :

```

    a      M  00 00 01 00 00 00 00 00 00 00 00 00 00 00
    C a      M  {0x0000,0x0001,0x0000,0x0000,0x0000}

```

For subtraction of pointers the compiler subtracts the pointers and returns the number of items to which they point between them. Look at this example:

```

int a[5];
int *ap,*bp,x;

void main()
{
    ap=a;
    bp=&a[4];

    x=bp-ap;

endit:
    while(1);
}

```

Try running this program and examine the values of ap, bp and x - at the end the value x is 4, even though the difference between ap and bp is actually 8 bytes.

```

    C ap      M  38 ->0x0000
    C bp      M  40 ->0x0000
    C x       M  0x0004

```

To subtract pointers to different types of objects (e.g. a pointer to a char and pointer to an int) would be meaningless and the compiler will generate an error if this is attempted.

Pointers can be compared and all the comparison operators (<,<=,==,!=,>,>=) can be used. The pointers must be pointing to the same type of object unless the == or != operators are used in which case they can point to different object types. Traditionally a null pointer (where the pointer value is 0) is used to indicate that the pointer does not point to a valid object, it may be used to indicate an error when a function returns.

11.6 Exercises

1. Look again at the string copy function test and run it line by line. Look at the value of the pointer "f" within the StringCopy function. You may like to look at the list file from the assembler (the LST file). Where in PIC ROM is the string "Test" held ? Can you see how the PIC C Compiler is marking strings held in ROM instead of RAM ?
2. Try to write your own versions of the following functions (which are all included in the strings library):

Add string s to the end of string d and return string d :

```
unsigned char *  
    strcat(unsigned char *d,unsigned char *s);
```

Find character c in string d and return a pointer to it if found. Return 0 if not found :

```
unsigned char *  
    strchr(unsigned char *d,unsigned char c);
```

Compare strings d and s and returns 0 if equal, -1 if d is less than s, and +1 if d is greater than s :

```
strcmp(unsigned char *d,unsigned char *s);
```

12 Structures & Unions

12.1 Introduction to Structures

A structure is a technique within C to group a number of related variables and form a new type. Consider a timer application where there are a number of alarm times each of which turns on, or turns off an output of PORTB of the PIC. Each alarm time has four items associated with it - the time in hours, the time in minutes, the bit number of PORTB to turn on or off, and a flag to show whether to turn the bit on or off. These could be represented by the following variables:

```
unsigned char Hours;      // Time for alarm (Hours)
unsigned char Mins;      // Time for alarm (Mins)
unsigned char Bit;       // Bit number (output) from 0 to 7
unsigned char Action;    // Action - 0 turn off, 1 turn on
```

The problem is that if we have 16 alarm times then we need to generate each of these items 16 times. We could do this with 4 arrays:

```
unsigned char Hours[16]; // Time for alarm (Hours)
unsigned char Mins[16];  // Time for alarm (Mins)
unsigned char Bit[16];   // Bit number (output) from 0 to 7
unsigned char Action[16]; // Action - 0 turn off, 1 turn on
```

However this is not very neat, C offers us a much better (and more efficient) way to group all the data items for an object together. This grouping is called a structure. Here is how to declare a structure which we shall call AlarmTime which holds the information we need:

```
struct AlarmTime
{
    unsigned char Hours;      // Time for alarm (Hours)
    unsigned char Mins;      // Time for alarm (Mins)
    unsigned char Bit;       // Bit number (output) from 0 to 7
    unsigned char Action;    // Action - 0 turn off, 1 turn on
};
```

Now "AlarmTime" is a new type keyword which we can use to declare a variable which holds all of the items in our structure. For example here is how to declare a single alarm time:

```
AlarmTime Alarm;
```

Now Alarm is a variable which holds all of the information in the structure. To access the variables which are members of the Alarm variable the dot operator (.) is used. For example this code will set the alarm time to midnight:

```
Alarm.Hours=0;          // Set Hours member of Alarm variable to 0
Alarm.Mins=0;           // Set Mins member of Alarm variable to 0
```

If the current time is represented by the variables Hours and Mins, then this code will check if the alarm time has been reached, and will turn the appropriate bit of PORTB on or off according to the Alarm structure:

It is possible to have structures within structures. Here is an example, continuing the timer theme, where we have one structure called Time which represents a time in hours and minutes, and the second an alarm time which includes the Bit and Action variables:

```
struct Time
{
    unsigned char Hours;
    unsigned char Mins;
};

struct AlarmTime
{
    Time ActionTime;      // Time at which action is to occur
```

```

    unsigned char Bit;          // Bit number (output) from 0 to 7
    unsigned char Action;       // Action - 0 turn off, 1 turn on
};

    Time CurrentTime;          // Current time
    AlarmTime Alarm;           // Time at which an alarm is triggered

```

Now the code to check if an alarm time has been reached will look like this:

```

    if (Hours==Alarm.ActionTime.Hours &&
        Mins==Alarm.ActionTime.Mins)    // Is alarm time reached ?
    {
        if (Alarm.Action)                // Test action to be undertaken on alarm
            PORTB|=(1<<Alarm.Bit);       // Turn bit on
        else
            PORTB&=~(1<<Alarm.Bit);       // Turn bit off
    }

```

It is quite easy to have an array of structures. So if we wanted up to 16 alarm timers then the array could be declared like this:

```

    AlarmTime Alarms[16];

```

To set the time of all alarms to midnight then this code could be used:

```

    unsigned char i;

    for(i=0; i<16; i++)
    {
        Alarms[i].ActionTime.Hours=0;
        Alarms[i].ActionTime.Mins=0;
    }

```

It is possible to assign a structure using =, in the same way as normal variables. In this case all the memory from one variable is copied to another, so all the variables in the two structures are identical. You do need to be careful using this form for C programs on the PC when handling memory allocation, however this is not an issue for WIZ-C Professional. Here is another way to set all times to midnight using assignment - this method is a little bit neater, but does use another variable:

```

    unsigned char i;
    Time Midnight;

    Midnight.Hours=Midnight.Mins=0;

    for(i=0; i<16; i++) Alarms[i].ActionTime=Midnight;

```

12.2 Pointers to structures

It is possible to have pointers to structures in the same way as to any type of variable. Here is how to declare a pointer to an AlarmTime structure:

```

    AlarmTime *atp;           // Pointer to structure AlarmTime

```

Now to set the time of an AlarmTime structure which is being pointed to by the pointer atp then the following can be used (this code sets it to midnight):

```

    (*atp).Hours=0;
    (*atp).Mins=0;

```

It is so common to use pointers to structures that the form "(*pointer)." can be abbreviated to the -> operator. Thus the example above can be rewritten as:

```

    atp->Hours=0;
    atp->Mins=0;

```

This can be very useful within functions which have to process structures. Here is an example function to check an alarm time which is passed as a pointer:

```
unsigned char CheckAlarm(AlarmTime *at)
{
    if (Hours==at->ActionTime.Hours &&
        Mins== at->ActionTime.Mins)    // Is alarm time reached ?
    {
        if (at->Action)                // Test action to be undertaken on alarm
            PORTB|=(1<<at->Bit);        // Turn bit on
        else
            PORTB&=~(1<<at->Bit);        // Turn bit off
        return 1;
    }
    return 0;
}
```

This function returns 1 if the alarm time was hit, and 0 if it was not. So now we can check every alarm time using pointers and the new function CheckAlarm:

```
AlarmTime *atp;                // Pointer to Alarms array
unsigned char i;

atp=Alarms;                    // Set atp to first item of Alarms array

for(i=0; i<16; i++)
{
    CheckAlarm(atp);            // Check if alarm time is met & action it
    atp++;                      // Now atp points to next item in array
}
```

12.3 Sub-Bit variables

It is possible using structures to pack variables which take a small number of bits into memory. This is done by using a colon (:) and then following it with the number of bits to be used by the variable. For example:

```
struct AlarmTime
{
    unsigned char Hours;
    unsigned char Mins;
    unsigned char Bit:3;        // Bit only takes values from 0 to 7
    unsigned char Action:1;     // Action is a one bit flag
};
```

This structure only takes 3 bytes in RAM instead of 4, but works exactly the same way as the previous structure. However within the PIC it is much slower than the previous version. The PIC supports single bit operations very efficiently so structures which only use single structures are actually quite efficient. However variables in structures which use 2 or more bits within that structure are quite slow in code terms and should be restricted to situations in which RAM code space is critical.

Here is an example of a Flags structure which can be used to set and test conditions:

```
struct Flags
{
    unsigned char GameRunning:1;
    unsigned char GameOver:1;
    unsigned char SecondPassed:1;
    unsigned char MinutePassed:1;
};
```

This structure only takes one byte in memory.

Please note, that owing to the operation of two's complement arithmetic, then single bit variables may expand to unexpected values, consider:


```

struct s
{
    char a:1;
};

s Test;
char x;

Test.a=1;
x=Test.a;

```

The single bit member a can take the binary values 0, or 1. If it takes the value 1 then because it is a signed value then is equivalent to -1. Therefore after this program is run the value of x is -1.

12.4 Unions

A union is similar to a structure, however all the variables occupy the same space:

```

union Joint
{
    char x;
    int y;
    long z;
};

```

The size of the union is 4 bytes (the length of the longest member), and changing any value will change all values within the union because they all occupy the same space. Unions are useful when a memory area changes its meaning according to the function of the program at the time.

Within a structure unions do not need to be named (they are "anonymous"). Consider:

```

struct example
{
    char a;
    union
    {
        char x;
        int y;
        long z;
    };
};

```

Now the members x,y and z can be addressed in the same way as any member of the structure, but are all in the same memory space - the size of this structure is 5 bytes within RAM.

12.5 Exercises

1. Work some of the examples shown above into WIZ-C Professional programs and test them. Do they work as expected ? You may need to examine structures as memory dumps - structures are held in memory as byte after byte.

13 The Pre-processor

The pre-processor is part of every C Compiler. It operates before the main C Compiler and controls compilation.

Any line which starts with a # is a pre-processor command - we have already seen the use of the #include pre-processor command which includes the entire contents of another file as if they were part of the current file. We'll take a look at the standard pre-processor commands here.

Pre-processor commands do not need to be followed by a semi-colon.

13.1 #include

The #include directive includes another file. It has two forms:

```
#include <filename>

#include "filename"
```

The first form (with the angular brackets) searches for the file in the standard directories used for the compiler. Within WIZ-C Professional the include files are included in a directory called "Libs". Including any of the library files or processor header files should be undertaken with the first form:

```
#include <P16F877.h>
#include <datelib.h>
```

The second form searches in the project directory first, and then within the standard directories. This form should be used for project include files which may include definitions of all the variables and functions used in the program.

```
#include "Variables.h"
#include "Functions.h"
```

Further examples of this are shown in the full example program.

13.2 Conditional compilation

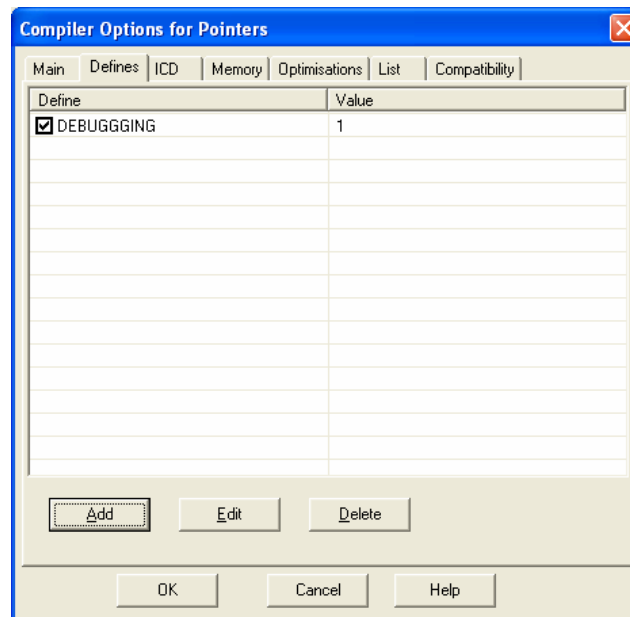
Conditional compilation allows the compiler to compile certain sections of code dependant on defines and values. For example:

```
void main()
{
    #ifdef DEBUGGING
        Wait(1);
    #else
        Wait(1000);
    #endif
}
```

The #ifdef directive tells the compiler to compile the following lines (before the #else, or before the #endif) if the symbol DEBUGGING has been defined. Symbols can be defined using #define:

```
#define DEBUGGING
```

Note in WIZ – C symbols can also be defined by using the Project | Current Project options menu option and then the debugging Tab :



Here the symbol is defined as 1, the tick box will define the symbol if selected and leave it undefined if not. This is really useful for setting project options which will apply to all files.

Now if DEBUGGING is defined the program will wait for 1mS at the program start, if it is not defined it will wait for 1 Second at the program start.

`#ifndef` is similar to `#ifdef`, but only includes the following code if the symbol is not defined.

`#if` is followed by a constant expression, for example:

```
const int DEBUGGING=1;

void main()
{
    #if DEBUGGING==1
        Wait(1);
    #else
        Wait(1000);
    #endif
}
```

This example is similar to the last one, but the `DEBUGGING` condition is set to 0 to 1 to change the delay.

#elif is equivalent to #else, #if. For example:

```
#if DEBUGGING==2
    Wait(10);
#elif DEBUGGING==1
    Wait(1);
#else
    Wait(1000);
#endif
```

In this example only one of the Wait statements will be included.

13.3 Macros

The `#define` directive defines a replacement for a symbol whenever it is found. For example:

```
#define WAIT 1000
```

Now whenever the symbol WAIT is found within the program it will be replaced by the string 1000. No replacement needs to be defined. Consider this code:

```
#ifndef MAIN
#define EXTERN
#else
#define EXTERN extern
#endif

EXTERN char x;
```

This code might be included in a header file which is included by two C files within a project. The first file might have the following:

```
#define MAIN
#include "Project.h"
```

The second file would have the following:

```
#include "Project.h"
```

Now when the first file includes the header file the symbol MAIN is defined, so the word EXTERN is replaced by nothing. So the variable x is defined and space is allocated for it. In the second file the symbol MAIN is not defined, so the symbol EXTERN will be replaced by extern and the variable x will be defined, but space will not be allocated for it. Without this type of operation an error will be generated as the variable x will be allocated twice.

Symbol replacements using #define can have arguments. For example:

```
#define CtoF(x) (x*9/5+32)
```

This macro converts Centigrade to Fahrenheit. Now whenever it is encountered in the program it will be replaced by this expression and x will be replaced by the value in the macro call. For example:

```
int TempC,TempF;

TempF=CtoF(TempC);
```

will be replaced by:

```
int TempC,TempF;

TempF=(TempC*9/5+32);
```

Note that this type of define (called a macro) looks like a function call, however it is not a function call as the entire code for the macro is inserted every time that the macro is used.

13.4 WIZ-C Professional specific pre-processor directives

The directive #pragma is used for C Compiler specific directives. For example:

```
#pragma stack 0x7f
```

This is a WIZ-C Professional specific directive to set the software stack pointer to the value 0x7f. The C reference chapter of the manual shows all the WIZ-C Professional specific directives.

14 Support for the PIC

14.1.1 Header files

There is a header file for every type of PIC supported by the WIZ-C Professional Compiler. The header files are called "P16nnnn.h" where nnnn is the processor type, e.g. P16F84.h. They may be included using #include:

```
#include <P16F84.h>
```

The header files check that the processor type matches the header file, an error will be generated if not.

As we have seen above the single file pic.h will include the correct header file for the current project options :

```
#include <pic.h>
```

The standard PIC file registers are all included in the header files under the same names as those shown in the PIC data books. The bits within the file registers are also included as enumerated constants. For example to enable and disable interrupts the following C Code may be used:

```
INTCON|= (1<<GIE);          // Enable interrupts
INTCON&=~(1<<GIE);         // Disable interrupts
```

In fact all the bits are also included by using the Microchip name preceded by a lower case b. These are bit variables (non-standard ANSI – please see the manual). So the example above can be recoded as :

```
bGIE=1;                     // Enable interrupts
bGIE=0;                     // Disable interrupts
```

This type of C code is translated into PIC BSF and BCF instructions, and is, therefore, very efficient.

14.1.2 Port Structure

Within the header file ports are defined as normal unsigned char types and as structures. The structures are named type is sPort, and the names are PA,PB,PC,PD and PE, or PG for the 8 pin devices. The bits are named B0 to B7.

The following code turns on bit 0 of PORTB, firstly by using the unsigned char variable PORTB, and secondly by using the structure:

```
PORTB|=1;                   // Turn on bit 0
PB.B0=1;                    // Turn on bit 0
```

The following code reads bit 4 of PORTC into variable x, returning 0 or 1 dependant on whether the bit is set or reset:

```
x=(PORTC&0x10)>0;          // Test bit 4 (return 0 or 1)
x=PC.B4;                   // Test bit 4 (return 0 or 1)
x=bRB4;                    // Test as a bit
```

15 Real Time Programming Example

Now we'll take a look at a full real time programming example. In this chapter we'll make use of PIC timers and interrupts and the FED library routines for driving an LCD display.

The full source is shown in Appendix A - Real Time Programming Source.

Use the project DigTimer.PC which can be found in the Projects\Learn To Use C\15 - DigTimer directory within the WIZ-C Professional installation.

The project that we'll develop is a fully featured timer with 8 outputs. It will have an LCD display showing the current time and which outputs are active. It will have up to 16 timer events, each of which will turn on or off a specified output pin. There will be 3 control buttons for the time - Up, Item and Set.

This project will run on the WIZ-C Professional 16F877 development board.

We'll develop the project incrementally - firstly we'll get a simple clock application running, then we'll include the timer events, finally we'll include the control buttons. The application will be debugged initially using the simulator, but then using the real hardware.

15.1 Application Designer Version

Note that this example can be simplified considerably using the application designer which has library elements for push buttons and LCD.

15.2 The PIC timer

All of the 14 bit core PIC's have at least one timer. The 16F877 has 3. In this example we'll use Timer 0 which is the timer that all PIC's have. Timer 0 is an 8 bit timer - it counts from 0 to 255 and then back to 0 again. It can be set to run from an external or the internal clock, and can be set with a pre-scalar which divides the clock source before it is used by timer 0. Timer 0 is controlled by a register called the Option register.

The OPTION_REG register is a readable and writable register which contains various control bits to configure the TMR0 prescaler/WDT postscaler (single assign-able register known also as the prescaler), the External INT Interrupt, TMR0, and the weak pull-ups on PORTB.

Option Register

bit			Timer Prescaler	Watchdog prescaler
bit 7:	RBP U: PORTB Pull-up Enable bit	1 = PORTB pull-ups are disabled	0 = PORTB pull-ups are enabled by individual port latch values	
bit 6:	INTEDG : Interrupt Edge Select	1 = Interrupt on rising edge of RB0/INT pin	0 = Interrupt on falling edge of RB0/INT pin	
bit 5:	T0CS : TMR0 Clock Source Select bit	1 = Transition on RA4/T0CKI pin	0 = Internal instruction cycle clock (CLKOUT)	
bit 4:	T0SE : TMR0 Source Edge Select bit	1 = Increment on high-to-low transition on RA4/T0CKI pin	0 = Increment on low-to-high transition on RA4/T0CKI pin	
bit 3:	PSA : Prescaler Assignment bit	1 = Prescaler is assigned to the WDT	0 = Prescaler is assigned to the Timer0 module	
bit 2-0:	PS2:PS0 : Prescaler Rate Select bits	000 001 010 011 100 101 110 111	1 : 2 1 : 4 1 : 8 1 : 16 1 : 32 1 : 64 1 : 128 1 : 256	1 : 1 1 : 2 1 : 4 1 : 8 1 : 16 1 : 32 1 : 64 1 : 128

So from this table we can select the value for the option register. We are not using interrupts and PORTB will be an output, so RBPU can be 1, INTEDG is irrelevant. T0CS needs to be 0 to use the internal clock. T0SE is irrelevant because the clock is internal. PSA should be set to 0 to assign the prescaler to Timer 0.

The 16F877 board has a 20MHz clock. Therefore the internal instruction cycle which is 1/4 of the main clock runs at 5Mhz or 200nS. We are going to count overflows of the Timer to keep time, however it doesn't matter what value we set the prescaler to, the timer will never overflow at a rate which divides exactly into one second.

We'll set the time to overflow at roughly once every millisecond. We do this by setting the prescaler to 32, now the timer will overflow every 1.6384mS. So every second will pass when we have counted 610 overflows of timer 0. This is not exact - there will be an error of 0.05% every second.

Every minute will pass when we have counted 36621 overflows of timer 0. This is still not exact - the error this time is 0.000256%. This is an error of 1.5 seconds per week which is more accurate than the crystal timing source.

Therefore we will hold two counters - one for seconds and one for minutes. The seconds counter will be slightly faster than the minute - it will reach the minute just before the minute counter. Therefore we'll reset the seconds counter when the minute counter reaches its own time and therefore keep the two counters in track.

This results in setting OPTION_REG to the binary value 0000 0100, or hex 4.

15.3 Stage 1 - Simple clock

We'll use a header file to hold the variables which apply to our project. Create a new directory and project called Timer (or use our example in Chap15). Create two new files in this directory - call one Timer.c and the other Timer.h. Insert Timer.c into the project window. Enter the following into the Timer.h file:

```

// #define DEBUGGING

#ifdef MAIN
#define EXTERN
#else
#define EXTERN extern
#endif

#ifdef DEBUGGING
const int SECONDS=1;           // Number of overflows per second
const int MINUTES=60;          // Number of overflows per second
#else
const int SECONDS=610;         // Number of overflows per second
const int MINUTES=36621;       // Number of overflows per second
#endif

EXTERN unsigned char Second;    // Current time
EXTERN unsigned char Minute;
EXTERN unsigned char Hour;
EXTERN unsigned char Day;

// Structures & enumerations
enum Flags {
    T0Overflow=1,               // Set when timer 0 has overflowed
    SecPassed=2,                // Set when one second has passed
    MinPassed=4                 // Set when one second has passed
};

EXTERN unsigned char Flag;

// Functions

void AddTime(unsigned char Fast); // Increment the time
void DoT0Overflow();              // Action whenever Timer 0 overflows
void TimerInit();                 // Initialise the system

// Variables

EXTERN unsigned int SecCount;     // Counts overflows for seconds
EXTERN unsigned int MinCount;    // Counts overflows for minutes

```

Note that we have the option in debugging to run the program about 600 times faster than normal for simulation purposes by deleting the `//` from the first line. The variable `Flag` is used to show when events occur, and the enumeration `Flags` defines the flags which make up this register - this should become clearer in the main program.

The following code should be entered into `Timer.c`, each function or block is shown followed by a description, all blocks need to be entered.

```

#include <P16F877.h>
#define MAIN
#include "Timer.h"

```

This section of code simply includes the headers for the 16F877 and also for the main timer application.

```

void main()
{
    TimerInit();           // Initialise the system

    while(1)               // Loop forever
    {
        if (Flag&T0Overflow) {DoT0Overflow(); Flag&=~T0Overflow;}
    }
}

```

This is the main loop. It initially calls the function `TimerInit` which initialises the timer. It then enters a continuous loop which checks for the occurrence of events and then calls the appropriate routine. For example the interrupt routine (which we'll look at later) sets a flag called `T0Overflow` in the `flags` variable when the timer overflows. Within the main loop we

check for this flag, and when it occurs we call the function DoT0Overflow(). We also clear the flag so that we only run the function once.

```
//
// Initialise the system
//
void TimerInit()
{
    Flag=0;                // Reset flags
    PORTB=0;               // Port B outputs all off
    TRISB=0;               // Drive all outputs of TRISB low
    OPTION_REG=4;          // Timer 0 internal, divide by 16
    TMR0=0;                // Clear timer 0
    Day=Hour=Minute=Second=0; // Set time to midnight, Sunday
    SecCount=MinCount=0;    // Clear counters
    INTCON=(1<<GIE)|(1<<T0IE); // Enable timer 0 interrupts
}
```

This is the TimerInit function. The variable Flag is cleared as are all the counters. The time is set to midnight on Sunday. The timer is set to operate internally and the prescaler set to divide by 16.

Now we want to set up an interrupt to occur when timer 0 overflows. Consult the PDF data file for the 16F877 on the CD ROM on which WIZ-C Professional is supplied. This describes the operation of interrupts in detail. In brief all interrupts are enabled if the GIE bit of the INTCON register is enabled. The Timer 0 interrupt is enabled if the T0IE bit of the INTCON register is set. The last line of the TimerInit() function enables interrupts when Timer 0 overflows.

```
//
// Interrupt handler
//
void Interrupt()
{
    if (INTCON&(1<<T0IF)) // Test Timer 0 overflow
        {INTCON&=~(1<<T0IF); Flag|=T0Overflow;}
}
```

This is the interrupt handler. Within WIZ-C Professional if a void function is defined called Interrupt() then it will be called automatically when an interrupt occurs. In this function we test the T0IF bit of INTCON (this bit is set when Timer 0 overflows). If it is set then we clear the bit, and then set the T0Overflow bit of the Flag variable. Now when the main loop tests the Flag variable it call the DoTimer0Overflow() function.

```
//
// Actions when timer 0 overflows
//
void DoT0Overflow()
{
    SecCount++;
    MinCount++;
    if (SecCount==SECONDS) {SecCount=0; AddTime(0);}
    if (MinCount==MINUTES) SecCount=MinCount=0;
}
```

This function is called when the timer overflows by the main loop. The seconds and minutes counters are incremented. If the seconds counter reaches its limit (defined in Timer.h - see above) then it is reset and the function AddTime() is called to add one to the seconds counter. If the minutes counter reaches its limit then the seconds and minutes counters are reset. The seconds counter is slightly fast and will always reach the minute first so the minute will have been incremented. All that is needed when the minutes counter reaches its limit is to clear both counters.

```
//
// Time has passed.
// Fast value is 0 if a second has passed
// The Fast value is set to 1 if minutes are to be increased
// The Fast value is set to 2 if hours are to be increased
```

```

// The Fast value is set to 3 if days are to be increased
// directly - this is used for setting the time.
//
void AddTime(unsigned char Fast)
{
    if (Fast) // Set the time
    {
        Second=0;
        if (Fast==1) Minute++; if (Minute==60) Minute=0;
        if (Fast==2) Hour++; if (Hour==24) Hour=0;
        if (Fast==3) Day++; if (Day==7) Day=0;
        return;
    }

    Second++;
    if (Second!=60) return;
    Second=0;
    Minute++;
    Flag|=MinPassed; // Show a minute has passed
    if (Minute!=60) return;
    Minute=0;
    Hour++;
    if (Hour!=24) return;
    Hour=0;
    Day++;
    if (Day==7) Day=0;
}

```

This function adds one to the time. If Fast is 0 then the seconds are incremented by one and Minutes, Hours and Days tested to see if they need to be increased. If Fast is 1 or greater then the time is being set, so minutes, hours or days will be immediately increased dependant on whether Fast is 1, 2 or 3 respectively

Compile this code. Add SecCount, MinCount, Second, Minute, Hour and Day as decimal watches (you will need to set SecCount & MinCount as Word watches). Trace the Second and Minute watches (right click the watch, use the Modify Watch option, click the trace button). Start the simulation running (you will need to set the Update Rate as fast as possible). Run it until about 65 Seconds (this may take up to 30 minutes dependant on the speed of your system. Examine Second and Minute using the wave analyser (examine each variable using the Add as 8 Line Traces button) - you will need to zoom out a lot. Once you can see up to the end of the trace zoom in around 60 seconds and measure the time between second changes - you should find they are 1.0S for all changes except from 59 to 60S which is 1.03S - this shows the effect of the seconds counter.

You will find that the times are not exactly aligned with the simulation time - this is owing to the time taken in the TimerInit() function.

15.4 Stage 2 - Add timer events

Now we'll add alarm timer events. We'll use a structure to describe a timer event. Add the following to the Timer.h file:

```

struct TimerEvent {
    unsigned char Day; // Day of event (has value 8 for every day)
    unsigned char Hour; // Hour of event
    unsigned char Minute; // Minute of event
    unsigned char BitAction; // Action and bit to set
};

EXTERN TimerEvent Timers[16]; // Array of timer events

```

This defines an array of timer events. The BitAction variable is set from 0 to 7 to turn off that bit of PORTB, or from 8 to 15 to turn on bits 0 to 7 respectively.

We don't need to initialise the timers - all memory is cleared to 0 when the PIC resets, so all timers will be set to Midnight on Sunday.

Add this function to the end of the Timer.C file, include its definition in Timer.h.

```
//
// Check if an event time has been reached
//

void CheckEvent()
{
    unsigned char i;
    TimerEvent *tep; // Pointer to timer event

    tep=Timers;

    for(i=16; i; i--) // Loop to check each timer
    {
        if (tep->Minute==Minute && tep->Hour==Hour &&
            (tep->Day==7 || tep->Day==Day))
        {
            unsigned char Mask; // Holds mask for bit

            Mask=1<<(tep->BitAction&7);
            if (tep->BitAction&8) PORTB|=Mask; // Turn on bit
            else PORTB&=~Mask; // Turn off bit
        }
        tep++; // Next timer event
    }
}
```

The check is undertaken with a pointer to the events. If the event matches the current time (when the day matches, or if the event day is 7 (which is every day)), then the required bit of PORTB is turned on or off. Note the form for the loop which is a very efficient form for WIZ-C Professional - this is explained in the optimisations section of the manual.

Change main as follows:

```
void main()
{
    TimerInit(); // Initialise the system

    Timers[0].Minute=2;
    Timers[0].Hour=0;
    Timers[0].Day=0;
    Timers[0].BitAction=1+8;
    Timers[1].Minute=3;
    Timers[1].Hour=0;
    Timers[1].Day=0;
    Timers[1].BitAction=1;

    while(1) // Loop forever
    {
        if (Flag&T0Overflow) {DoT0Overflow(); Flag&=~T0Overflow;}
        if (Flag&MinPassed) {CheckEvent(); Flag&=~MinPassed;}
    }
}
```

This defines two timer events - one at 2 minutes past midnight and one at 3 minutes past to turn on and then off bit 2 of PORTB.

Go to the Timer.h file and comment out the DEBUGGING line so that it reads:

```
#define DEBUGGING
```

This will considerably increase the speed at which minutes and hours are incremented. Run the simulation and watch Port B go to 2 and then back to 0 as minutes increments to 2 and then 3. Experiment with the initial values set in main() to check that the timers work for - you can modify Day and Hour by double clicking them in the watch window to avoid waiting long periods for the simulation to run.

15.5 Stage 3 - The display and HMI

15.5.1 Connecting the display and buttons

Now we can set up the display and press buttons to provide the input and output to the user. The system will work on a menu operated basis.

There will be 3 buttons - "Set", "Item" and "Up", these are connected to bits 7, 6 and 5 of PORT D respectively.

Pressing the Set button will enter setting mode, each press will move on to the next object which can be set - there are 16 events and the time which can be set. When Set is pressed once Event 1 will be displayed and set, when pressed again Event 2 and so on up until event 16, then pressing Set will move on to setting the time.

The Item button will move to the next item. When Set is pressed once then Event 1 may be set, the Item button will move from setting Day, to Hour, to Minute and then the action (bit number and turn on or off) for this event. For setting the time only Day, Hour and Minute may be set - the clock will reset seconds to 0 as soon as minute is set.

Finally the Up button will move on the time or action. When pressed the current Item will increment by one.

All buttons will auto repeat if held down for more than 1/2 second, and all buttons will be debounced in software.

When setting an event or the time, if buttons are not pressed for 30 seconds then the display returns to showing the time and state of the outputs.

Look at the circuit diagram of the F877 board. Note that the upper 4 bits of port D are connected to the key pad port by 4K7 resistors. As we will only use 3 buttons we can simply connect the buttons between the keypad port and ground. (A keypad requires a scanner to read columns and rows). Reading port D will now read the buttons. We'll connect the Set button to bit 7, the Item button to bit 6, and the Up button to bit 5, the buttons are connected between the keypad pin and earth (keypad pins 1, 2 and 3 respectively).

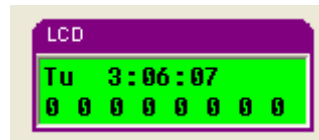
The LCD display is connected pin for pin to CONN2. Pin 1 of CONN2 to pin 1 of the LCD display, pin 2 of CONN2 to LCD module pin 2, and so on up until pin 14. Nearly all LCD modules conform to the same pinout and protocols.

This is actually as follows :

LCD Connection	PIC Pin
RB4	PORT D : 4
RB5	PORT D : 5
RB6	PORT D : 6
RB7	PORT D : 7
E	PORT E : 1
RS	PORT D : 2
RW	PORT D : 3

We will only have one display routine which will update the LCD display and show exactly what is required dependant on the current state of the menu variables. This routine can then be called from anywhere in the main program to update the display whenever information has changed. There will be two variables controlling the state of the menus, these variables are called "menu" and "item".

The screen shot below shows the LCD showing the time and state of all outputs during normal operation :



15.5.2 Menu variable

The Menu variable defines the current item displayed and modified by the buttons:

Value of menu	State	Display
0xff	Normal	Upper line - time and day Lower line - active outputs
Between 0 and 15 inclusive	Setting an event (when Menu is 0 then Event 1 will be displayed)	Upper line - time and day Lower line - Time and action for event
16	Setting the time	Upper line - time and day

If menu is not 0xff then a timer called "MenuTimeOut" will be running (it is decremented every second) - this is reset to 30 every time that a key is pressed, however if it reaches 0 then the state of menu is reset to 0xff. I.e. if keys are not pressed then eventually menu will return to 0.

To change the value of menu the "Set" button will be pressed.

15.5.3 Item variable

Value of item	State	Display
0	Setting Day	Change Day when "Set" button is pressed
1	Setting Hour	Change Hour when "Set" button is pressed
2	Setting Minute	Change Minute when "Set" button is pressed
3	Setting Bit Number and action	Change Bit # when "Set" button is pressed, the display will show bit numbers 0 to 7 whilst the action is 0 (turn off) then 0 to 7 whilst the action is 1 (turn on), and then repeats.

To change the value of item the "Item" button will be pressed. To change a value the "Up" button is pressed.

The Item button has no effect when "Menu" is 0xff.

15.5.4 Source code

The following section shows the new code required for the display and keypad. Note that a large number of changes are made and so those intending to enter the code are recommended to copy the completed example shown in Appendix A.

The following variables are added to Timer.H. (Functions shown below must also be added).

```

EXTERN unsigned char Menu;
EXTERN unsigned char Item;
EXTERN unsigned char MenuTimeOut;

```

These variables represent the current Menu display, the item being set, and the Menu timer which is decremented every time a second passes without pressing a key. When it reaches 0 the menu returns to showing the time.

```

enum Keys {
    NO_KEY=0xff,          // No button is pressed
    SET_KEY,             // Enter button is pressed
    ITEM_KEY,            // Next button is pressed
    UP_KEY               // Up button is pressed
};

EXTERN unsigned char Key;          // Last key pressed
EXTERN unsigned int KeyDelay;      // Delay before key repeat

```

The enumeration shows values for keypresses and represent the buttons being pressed.

A new flag value is created:

```

enum Flags {
    T0Overflow=1,          // Set when timer 0 has overflowed
    SecPassed=2,           // Set when one second has passed
    MinPassed=4,           // Set when one second has passed
    KeyPressed=8           // A button has been pressed
};

```

The KeyPressed flag is set when a new button press is detected.

Now create a new file (**File | New**), save it as "TimerHMI.C" in the timer directory. Add it to the project (**Project | Add/Insert Item**) and enter the functions and code shown below - as before functions are described after each code block.

```

#include <P16F877.h>
#include <Displays.h>
#include <Delays.h>
#include <Strings.h>
#include "Timer.h"

#define LCDPrintAt(x,y) LCD(0x100+0x80+x+y*0x40)
#define LCDOnOff(display,cursor,blink) LCD(0x108+display*4+cursor*2+blink)
#define LCDShift(cursor,right) LCD(0x110+(!cursor)*8+right*4)

const int LCDPORT=&PORTD; // Connect module to port D
const int LCDEPORT=&PORTE;
const int LCDEBIT=1;
const int LCDSPORT=&PORTD;
const int LCDRSBIT=2;
const int LCDRWPORT=&PORTD;
const int LCDRWBIT=3;

const char *Days[]=          // Array of string representing days
{
    "Su",
    "Mo",
    "Tu",
    "We",
    "Th",
    "Fr",
    "Sa",
    "Ev"
};

```

This is the header block for the display routines. Note the headers included - these are used for various library routine. The LCD constants define the connections of the LCD port for the F877 development board. The constant Days array represents the days as strings - Day 1

which is Monday will be displayed as "Mo". Note the macros which send the correct character values to the display to set the print position, turn it off or on, and shift it left or right. These are not all used in this program.

The library function LCD() sends a single character to the screen. Character values greater than 0x100 hex are control characters which affect the display as shown.

```
void DisplayInit()
{
    Menu=0xff;           // Initial value of menu
    MenuTimeOut=0;       // Delay before returning menu to std display
    Item=0;              // Item - hours, mins, day, bit, event
    LCD(-2);             // Initialise display to 2 lines
    LCDClear();          // clear display
    LCDString("Timer");  // Start up string
    #ifndef DEBUGGING
        Wait(1000);      // Delay 1 s before starting
    #endif
}
```

This function is called from the main initialisation routine. It initialises the display and menu system. The library function "LCD(-2)" initialises the display with 2 lines. The Library function "LCDString()" displays the supplied character string on the display. This function displays the word "Timer" for one second before starting the main applicaiton.

The next function is the display function, it is called whenever the display is to be updated. It takes account of the state of the menu and displays the correct output, time or event. We'll look at this function in sections.

```
//
// Display values on screen
//
void Display()
{
    char ws[17];          // String to hold display values
    unsigned char i;

    LCDClear();           // Clear display

    if (Menu==0xff)       // Normal display (time & State of outputs)
    {
        LCDPrintAt(0,0); // 1st Line
        TimeString(ws,Hour,Minute,Second,Day,1); // Show the time
        LCDString(ws);
        LCDPrintAt(0,1); // 2nd line
        for(i=0; i<8; i++) // Show state of all outputs
            {LCD((PORTB>>i)&1+'0'); LCD(' ');} // Display 1 or 0 for output
        return;
    }
}
```

The first part of the function defines variables to be used within the function. ws is a useful character string and i is used as an index variable. When Menu is 0xff the display shows the current day, time and output state. Each output is shown as a 1 or a 0 followed by a space. For example to show that bits 2 and 7 are turned on whilst all others are turned off at 19:43 and 20 seconds on Thursday the following display will be shown:

```
Th 19:43:20
0 0 1 0 0 0 0 1
```

The function TimeString is shown further below. It prints the time to a string, the last parameter is set to 1 to display Seconds (it is not used within events).

```
if (Menu<=15)           // Setting an event
{
    LCDPrintAt(0,0); LCDString("Set Ev "); // Display "Set Ev"
    cPrtString(ws,Menu+1); LCDString(ws); // Display event number

    LCDPrintAt(0,1);
    TimeString(ws,Timers[Menu].Hour,Timers[Menu].Minute,0,
               Timers[Menu].Day,0); // Show the time
}
```

```

LCDString(ws); LCD(' ');

LCD(((Timers[Menu].BitAction&7)+'0'); LCD(' '); // Display event number

if (Timers[Menu].BitAction&8) LCD('1');          // Action is to set bit
else LCD('0');

LCDPrintAt(10,0);          // Print position for item
}

```

This part of the display function displays an event. The event number is defined by the Menu variable, but a 1 is added when the event number is shown. For example is displaying Event 1 set to 20:00 hours every day to turn on output bit 5.

```

Set Ev 1 D
Ev 20:00 5 1

```

The letter D shows that day is being set, this will change to H, M and B when hours, minutes and bit/action are being set. Our code block simply displays the Event number, day, hour and action. We'll show the action later on in the function.

```

if (Menu==16)          // Set the time
{
    LCDPrintAt(0,0); LCDString("Set Time "); // Display "Set Time"

    LCDPrintAt(0,1);
    TimeString(ws,Hour,Minute,0,Day,1);    // Show the time

    LCDString(ws);
    LCDPrintAt(9,0);          // Print position for item
}

```

This part of the function shows the display for setting the time. An example is:

```

Set Time D
Th 19:50:00

```

The seconds will always display as 0, again the Item is shown last on the display.

```

// Now print the item that we are setting
switch(Item)
{
    case 0: i='D'; break;
    case 1: i='H'; break;
    case 2: i='M'; break;
    case 3: i='B'; break;
}
LCD(i);          // Print the item
}

```

Finally this part of the display function shows the letter corresponding to the item being set.

```

//
// Print a time to a string
//

char *TimeString(char *ws,unsigned char Hour,unsigned char Min,
                 unsigned char Sec,unsigned char Day,unsigned char ShowSec)
{
    unsigned char at=0;

    strcpy(ws,Days[Day]);          // Print day to string
    ws[2]=' '; at=3;              // Space (gap)
    if (Hour<10) ws[at++]=' ';    // Hour leading space
    cPrtString(ws+at,Hour);        // Hours
    ws[5]=':'; at=6;              // Colon
    if (Min<10) ws[at++]='0';      // Minute leading zero
    cPrtString(ws+at,Min);         // Minute
    ws[8]=0;                      // End of string
    if (ShowSec)                  // Display seconds ?

```



```

{
    ws[8]=':': at=9;
    if (Sec<10) ws[at++]='0';          // Minute leading zero
    cPrtString(ws+at,Sec);
}
ws[11]=0;                            // End of string marker
return ws;
}

```

Here is the function which prints a time to a string supplied as Day, Hour, Minute and Second. The Seconds part is only printed if the ShowSec variable is 1. The library function cPrtString prints a number to a string. Note how leading zeroes are printed for hours, minutes and seconds as this is not performed by cPrtString.

Now we'll take a look at reading buttons. As buttons tend to bounce they must be sampled at a low rate. We'll use a function called ReadBut() to read the buttons and set the KeyPressed flag. This will be called every 50mS or so by adding a call to the DoT0Overflow() function:

```

void DoT0Overflow()
{
    SecCount++;
    MinCount++;
    if (!(SecCount&SECOF)) ReadBut();    // Read button every 50mS
}

```

To enable the program to work in debugging mode we use the following lines in timer.h:

```

#ifdef DEBUGGING
    const int SECOF=1;
#else
    const int SECOF=0x1f;
#endif

```

In normal operation Timer 0 overflows every 1.6mS. When the bottom 5 bits of SecCount are 0 then ReadBut() is called, this will happen every 32 overflows, or roughly every 51mS. Here is the ReadBut function:

```

//
// Read a button on a timer interrupt
//

void ReadBut()
{
    unsigned char NewKey=NO_KEY;

    TRISD|=0xf0;                    // Upper 4 bits of PORT D to read
    if (!(PORTD&0x80)) NewKey=SET_KEY;    // Set key
    if (!(PORTD&0x40)) NewKey=ITEM_KEY;    // Item key
    if (!(PORTD&0x20)) NewKey=UP_KEY;      // Up key

    if (NewKey==NO_KEY) {Key=NO_KEY; return;}    // No key press

    if (NewKey!=Key)
    {
        KeyDelay=INITKDELAY;
        StateFlag|=KeyPressed;
        Key=NewKey;
        return;
    }

    if (!KeyDelay)
    {
        KeyDelay=REPKDELAY;
        StateFlag|=KeyPressed;
        return;
    }

    KeyDelay--;
}

```

Firstly port D is set to read, then it is read and checked to see if the button bits are low. If any bit is low then the NewKey variable is set to the button number. Note also that there is no

checking for 2 or more buttons pressed together - in this case the lower bit number will "win". If there is no key detected (all 3 bits high) then the function simply returns.

The variable Key is set to the value of the last button press detected. If Key is different from NewKey then we have a new key press. So the KeyDelay variable (which counts down for key repeats) is set to the constant INITKDELAY. The KeyPressed flag is set to show a new key has been detected, and Key is set to the new key value detected.

If Key and NewKey are the same then the key is held down. If the KeyDelay variable has reached zero then the key repeats - the KeyPressed flag is set again and the KeyDelay variable is set to the repeat value.

Finally if the button is held down (Key and NewKey are the same) and KeyDelay has not reached 0 then it is simply decremented by 1.

We need to add 2 new constants to Timer.h:

```
const int INITKDELAY=SECONDS/20/2; // Delay of 1/2S before 1st key repeat
const int REPKDELAY=SECONDS/20/5;  // Delay of 1/5S between key repeats
```

Recall that ReadBut is called 20 times a second and this is used to set the initial delay before key repeat to 1/2 seconds and a repeat rate of 5 times per second.

We have now read a button, but we must deal with it. This is handled within the main loop:

```
void main()
{
    TimerInit();           // Initialise the system
    DisplayInit();

    while(1)               // Loop forever
    {
        if (StateFlag&T0Overflow)
            {DoT0Overflow(); StateFlag&=~T0Overflow;}

        if (StateFlag&MinPassed)
            {CheckEvent(); StateFlag&=~MinPassed;}

        if (StateFlag&KeyPressed)
            {DoKeyPress(); StateFlag&=~KeyPressed;}
    }
}
```

We call the function DoKeyPress when a button press is detected and then clear the flag which shows a key has been pressed. Here is the function DoKeyPress:

```
//
// Handle a key press
//

void DoKeyPress()
{
    if (Key==SET_KEY)
    {
        Menu++;
        if (Menu==17) Menu=0xff;
        Item=0;
    }

    if (Key==ITEM_KEY)
    {
        Item++;
        if (Menu<=15)
        {if (Item==4) Item=0;}
        else
        {if (Item==3) Item=0;}
    }

    if (Key==UP_KEY)
    {
```

```

if (Menu<=15)
switch(Item)
{
case 0 : Timers[Menu].Day++;
        if (Timers[Menu].Day==8) Timers[Menu].Day=0;
        break;
case 1 : Timers[Menu].Hour++;
        if (Timers[Menu].Hour==24) Timers[Menu].Hour=0;
        break;
case 2 : Timers[Menu].Minute++;
        if (Timers[Menu].Minute==60) Timers[Menu].Minute=0;
        break;
case 3 : Timers[Menu].BitAction++;
        break;
}
if (Menu==16)
switch(Item)
{
case 0 : Day++;    if (Day==7) Day=0; break;
case 1 : Hour++;   if (Hour==24) Hour=0; break;
case 2 : Minute++; if (Minute==60) Minute=0;
        Second=0;
        break;
}
}
Display();
MenuTimeOut=NOACTMENUTO;
}

```

We check which button has been pressed.

If it is "Set" then the Menu variable is incremented by one (if it reaches 17 then it is reset to 0 - normal time). Item is set to 0 every time that Menu changes.

If it is "Item" then the Item variable is incremented. Here we need to check if it has reached 3 or 4 depending on whether an event or the time is being set, and then reset it to 0.

Finally if it is "Up" then a value is changed. If Menu is 0xff then nothing happens. If it is pointing to an event (i.e. Menu is between 0 and 15 inclusive) then the Day, Hour, Minute or Action of the event are incremented. If the time is being set then the Day, Hour or Minute is incremented depending on Item. In all cases if the item reaches the maximum value (say 24 for the Hour value) then it is reset to 0.

After changing the value the display is updated and the menu timer is restarted.

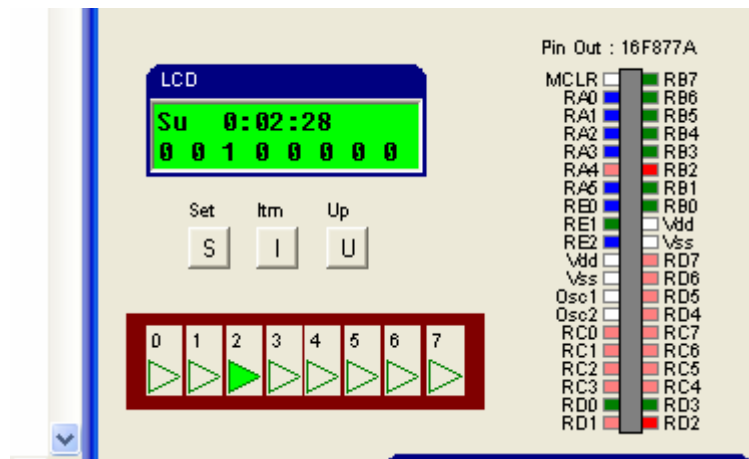
We have now completed the application. There are a few items to mop up such as including all these new functions in the header, and dealing with the menu timeout, however you can see these in the full code listing in Appendix A.

15.6 Simulating using external devices

We recommend that you get the full application operating from Appendix A before undertaking these exercises. All of this is included in the example directories.

This is an ideal project for simulation using the LCD and push button devices. Connect the LCD as shown above, and three push buttons which can be called Set, Item and Up to Port D bits 7, 6 and 5. They will need to connect common to ground.

The external devices we arranged in the example project like this :



The array of LED's has been set up and grouped with a rectangular background. In this example output 2 has been triggered.

You can run this program and use the push buttons to test it. Note that the program will not respond until around 1 second and then it will run quite slowly owing to the crystal frequency. You will need to push the button for a short time for it to register, and equally release for a short time to allow the button up event to be noted.

15.7 Further notes & Exercises

This program can be blown into an F877 on the development board. The buttons do not need to be connected - a flying lead from ground can be used to simulate button presses.

1. Add a power off warning so that the display flashes after power down until the time is set.
2. Allow the events to turn on or off between 1 and 8 outputs together. You will need to change the TimerEvent structure. Try to achieve this without increasing the size of a TimerEvent structure - you may wish to have events 1 to 8 turn on outputs and events 9 to 16 turn off outputs so that an event structure no longer holds the action, just a pattern of bits.
3. The crystal operates at 20MHz. In practice crystals are very accurate over time, but may not be absolutely accurate. For example our crystal may oscillate 20,000,123 times a second although it will stick very closely to this value. Normally this is corrected by using a variable capacitor to load the crystal. However we can correct it in software.

Add an additional function to our clock to change the value of the MINUTES variable to allow the time to be closely adjusted. It will need to be made a variable, not a constant. Very simply this could be a value which can be changed, a more complex procedure would allow the user to enter the error of the clock over a defined period (say one day) and correct the variable automatically.

4. Check the help file to see how to use the EEPROM data library functions. Save the time automatically to EEPROM every minute, and also save all the timer event values and the current value of PORTB. On reset read the values back from EEPROM. This will allow the timer to handle short power cuts without too much disruption.

How long will the EEPROM last if it has an endurance of 10 million write operations and it is written once a minute ? How could EEPROM life be extended ?

5. A real time operating system (RTOS) will run a number of threads simultaneously, chopping between threads at regular intervals. A RTOS is very useful because really slow functions can be run together with functions which need to be run at frequent intervals (such as DoT0Overflow()). Unfortunately an RTOS is very hard with these PIC's owing to

the operation of the stack.

The limiting factor of a system such as this is whether all actions which can happen between two timer 0 overflows can be completed - otherwise a timer overflow may be missed. The worst case in this program is when CheckEvent is called with all events set to the same time which is the current time.

Simulate this occurrence (you'll need to comment out calls to the display initialisation and routines to use the simulator). How long does it take to check all events ? What is the margin left before the next timer overflow ? Try reducing clock frequency - what is the minimum clock frequency you can use before missing overflows ? How could you stop this problem by changing the program (Hint - only look at one timer event at a time) ?

6. Connect a 64R Loudspeaker via a buffer transistor to a spare output and play a tone when an event occurs *without* affecting Timer 0 overflow detection.

Really advanced ! Play a tune on an event.

7. Consult the 16F877 data sheet and look at how Timer 1 and the Capture/Compare registers and interrupts operate. Can you modify the application so that the Seconds counter times out exactly once every second and there is no need for the minutes counter. (Hint - set a compare event every 10,000 clocks for an exact 2mS interrupt).

16 WIZ C Professional – Application Designer

16.1 Application Designer

So far we have programmed in C without using a number of the more powerful facilities of WIZ – C Professional. In this final chapter we'll use the Application designer which can allow programs to be developed very rapidly by quick incorporation of library functions and configuration of devices.

These projects are not provided as examples as the build up helps to understand the use of the Application designer.

16.2 Application Designer example Project #1 - Switches, LED's and a serial interface.

In the example project we will look at the development and simulation of a complete program using WIZ-C.

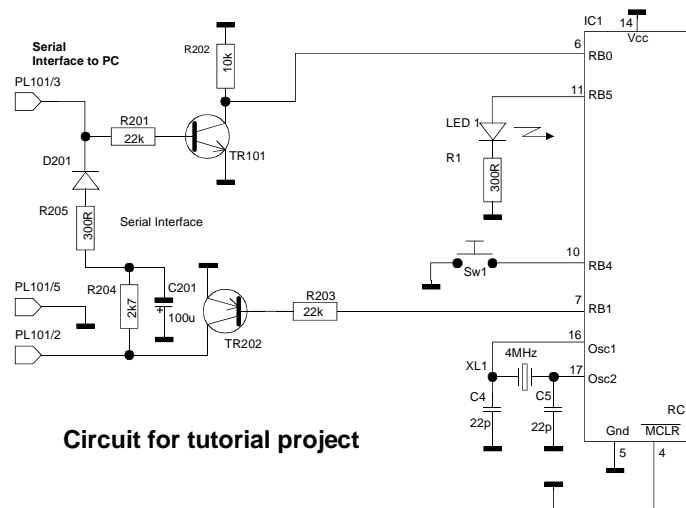
The program we will look at is designed for a 16F84 processor. The application will undertake the following simple functionality:

- 1) It will have a serial interface which may be connected to a standard PC using a 9 pin socket (PL101).
- 2) It will have a push button switch
- 3) It will have an LED

When a byte is received on the serial interface the LED will illuminate. Now when the push button is pressed the received byte will be sent back on the serial interface and the LED will be extinguished.

We will simulate this device using the real device simulation capability of WIZ-C.

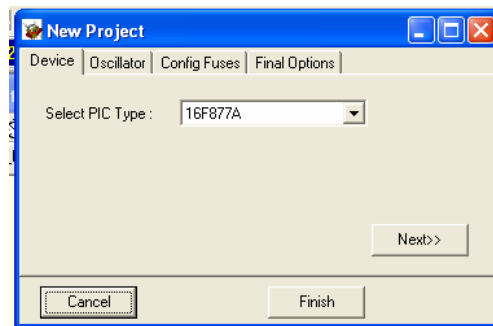
The circuit diagram is shown below:



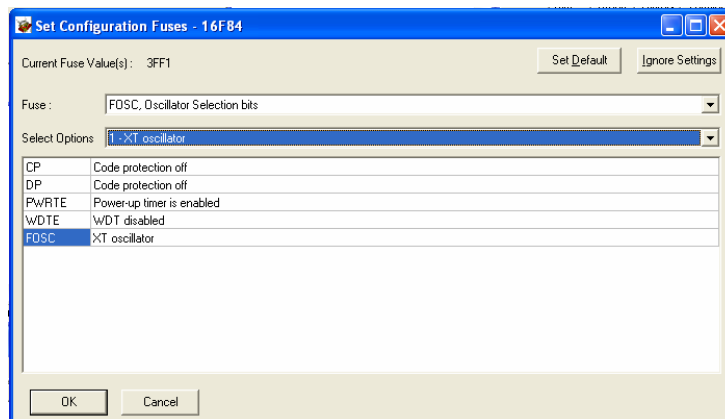
Note that by default WIZ-C will set Port B pull ups to enabled so there is no need to use a pull up on the switch input.

16.2.1 Opening a new project

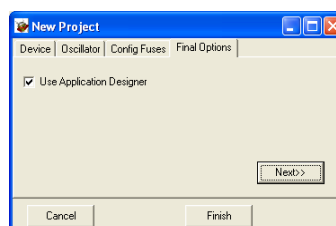
To open the new project then use the *Project | New Project or Project Group...* menu item. This brings up the new project wizard :



Use the “Select PIC Type” box to select the 16F84 device. Click the next button. This will bring up a number of standard oscillator frequencies – select 4MHz. Click Next, this brings up a button allowing the configuration fuses to be set, click the button to see the options. This is how we set it :



Click OK and then Next to display the final options. By default we would like to use the Application Designer so leave this option checked :



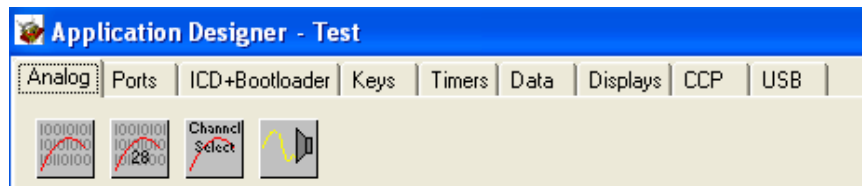
Finally press Next for the last time (or Finish).

A File dialog box will be brought up. Select the directory “C:\Program Files\FED\PIXIE\Projects\Tutor”. If the directory does not exist create it using the New Folder button on the Open Dialog. Enter the file name "Tutor", click Open. As this is a new project it may appear in minimal form, maximise the window by using the button in the top right of the window bar and then use the **Window | Arrange for Edit** option to position the new project on screen (you could also press ALT+E).

The top window will be the application designer. Check that the PIC type is 16F84 and ensure that the oscillator frequency shows 4000000 which is 4MHz. Alternate PIC types can be selected by clicking the button labelled "Change Device" (under the element groups), the

oscillator frequency may be selected from a list, or the exact value may be typed into the box. The oscillator frequency is entered in Hertz.

Look at the application designer window. The application designer holds software elements in groups at the top of the window :



A software *Element* is a library subroutine, or software component, which may be used within an application. The application designer allows software elements to be selected for use within the current application. The software elements are grouped by type. The element is the small square icon.

The application designer can be shown or hidden using the Show button on the tool bar :



16.2.2 Using elements within the application

The first element that we will select is the serial interface. There are 3 asynchronous serial interface elements all under the Data tab. Select the Data tab and hover over an element with the mouse - a small help box will appear with the element name. Select the element called "Software serial interface - Interrupt driven" by clicking it. The element icon looks like this:



Now right click the element and a pop up menu will appear. Select the menu option "Help on selected element" and read through the help file entry for this element. It probably won't all make sense at the moment.

Now you can add this element to the project by double clicking it, or by dragging it on to the picture of the PIC. Do this and the element will appear in the element store at the bottom of the application designer. A picture of the element in a box will appear on the drawing of the PIC. Note also that another element has also been included - this is the PORT B Driver element which is used by the serial interface and has been *hooked* into the project by the serial interface.

Note that the element will have connected one of its pins to the PIC - ISerialRx. This is because this element uses the PORT B interrupt input. The output of the element can be connected anywhere. Click the other pin of the element – if it is left for a few seconds a help hint will pop up displaying the pin name "SerialTx". Now click pin RB1 of the PIC to connect it to the element.

Now we must set the parameters of the serial interface. Click the parameters tab and the only parameter for this element will be displayed - "Serial Bit Rate". This will be set to 9600 - the default, leave it set to 9600.

Some software elements including the Software serial interface element allow the user to define software functions to be called automatically when events occur. An event is described in the Applications designer as an *Occurrence*. In this case the Occurrence is that a byte has been received on the serial interface. When a byte has been received we would like to illuminate the LED. Click the Occurrences tab and a list of occurrences will be displayed, in

this case there is only one occurrence "IRx". When a byte is received we would like to call a C function called "LEDOn".

Click the occurrence IRx to select it. In the "Calls for Occurrence" box type LEDOn and then click Add. The routine LEDOn will now be shown in the list of calls for this occurrence. Further functions could be added here if wished.

This completes the initialisation for the interrupt driven serial element.

Now we can add the switch which will be connected to pin B4. Select the Keys tab and double click the "Simple Switch" icon:



Again you will notice that two elements have been added, the first being the switch, the second being timer 0. The reason for this is that the simple switch includes debouncing and auto-repeat functions which need timer functions. The simple switch has automatically *hooked* in timer 0 as it uses it for the timer functions. Timer 0 can still be used by the application. You will notice that this element has the title "Simple Switch 0" in the blue title bar, this is because we can have multiple copies of the Simple Switch, the next would be called "Simple Switch 1".

Now connect the only pin on the element to pin RB4. The name of the pin will be SBIn0, this is not very meaningful so select the pin by clicking it. Now in the Pin Name box at the top of the application designer type the name "TxSwitch". Now when the application is generated there will be three symbols defined – a bit variable called TxSwitch and also two constants - TxSwitchPort and TxSwitchBit which represent the port and bit to which the switch is connected - we can test the switch directly by using a line such as:

```
if (TxSwitchPort&(1<<TxSwitchBit))
```

alternatively we can test it using the bit variable which is created with the same name :

```
if (TxSwitch)
```

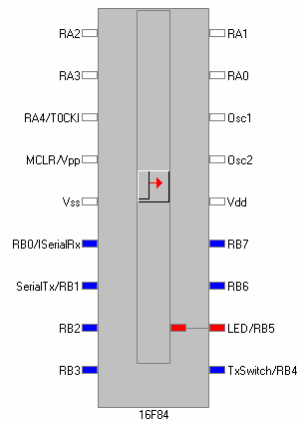
Examine the parameters of the switch - we don't need to change any of the default values, so now click the Occurrences tab. When the switch is pressed we would like to transmit the last received value. So enter a function name "TxLastRx" and add it to the list of calls for the occurrence SBPress0.

Now we can add the LED output which is on pin B5. Click the Ports Tab at the top and add the "Port Driver" element:



Connect the port output to pin RB5 and name it "LED". Initially the LED will be off so under the parameters tab select the initial value 0.

We have now completed the work with the application designer - our application will include initialisation, code and data for all the main functions of the application. Examine the PIC - it should look like the picture below, don't worry about the order of names on the pins of the PIC as it is dependent on the order of element selection.



You can print the PIC by right clicking the PIC graphic and using the *Print PIC* option of the pop menu. You can copy a picture of the PIC to the clipboard and paste into other applications by using the *Copy PIC* option of the same menu.

16.2.3 Generating the application for the first time

We have now selected the initial set of elements for the application, and the parameters, inputs, outputs, and occurrence calls have been defined, the application may be generated. To do this right click the PIC graphic and use the menu option *Generate Application* or press Control and the F9 key, or use the small button at the top left of the application designer:



The project window on the top right will show three files and a box titled "Compiler Options" will be shown, this allows the C Compiler options to be set. This box will appear the first time that a project is compiled, but will not appear again. To set the options after the first time then use the *Project / Set options for this project* menu option.

As we don't need to alter the default options then this can be ignored so click OK and the project will compile and assemble. At present we still have some code to define.

Now we would like to add some application specific code to the project. We would like an application specific header for the project. Click *File / New* and a new file will appear. Use the menu option *File / Save As* and select the file type Header Files, enter the filename "Tutor" and the file will be saved as "Tutor.h".

This file will include the memory variables used by our application. In this case it will simply be a flag which is set to 1 when a byte is received. Enter the following into Tutor.h:

```
bit RxFlag;           // Flag when byte is received
```

Double click the file in the project window called Tutor_User.C to open it. This file is produced automatically the first time that an application is generated (and is not generated again after that). We need to add the header to the file and also need to write the sub-routines which we defined for the occurrences. Move to the top of the file, you will notice that the first line of the file includes a header file called Tutor_Auto.h, this is the header which includes all the application designer information as well as the processor header. It should be included at the top of any additional files you include in the project. Include our new header by entering the following on the second line of the file:

```
#include "Tutor.h"
```

Look down the file to the UserInitialise function which will be empty, add the following code to the function between the curly brackets:

```
RxFlag=0;
```

This clears the receive flag while the program initialises. This isn't strictly necessary as the compiler clears memory when it runs, but is good programming practice and makes it clear what we are doing.

Examine the file. There is a function called UserLoop. This label will be jumped to by WIZ-C as it runs round its own main loop checking for occurrences and calling associated sub-routines. In this case we do not need to do any processing in the main loop - it is all undertaken by WIZ-C, so leave this section as it is.

Now look at the bottom of this file, the two functions that should be called when an occurrence happens will have been entered as templates. The LEDOn function will turn the LED on when a byte is received and will set the RxFlag when a byte is received, edit the function to read:

```
void LEDOn()
{
    RxFlag=1;           // Show a byte has been received
    LED=1;              // Turn on the LED
}
```

Note that when we named the output connected to our LED (recall that we called it "LED"), then the application generator automatically created the bit variable LED for us.

For the second routine which transmits the byte received we need to know what are the calls and variables used by the serial interface element. Position the cursor on the blank line in the middle of the TxLastRx function and press ALT and Enter together. A sub menu will appear, select the element Calls/Vars option, now a list of available functions for the elements used will appear. Select the pSerialOut line and double click it (or press Enter). A blank call will appear - we could have typed this in by hand, but this feature shows the parameters and some information on the call.

Bring up the Application Designer by pressing the extreme right hand button on the tool bar:



or by using the *Window | Application Designer* menu option. Click the serial interface element in the element store at the bottom of the Application Designer and click the Interface tab to see a list of interfaces for this element. This information is used to help finish the second routine which will transmit the last received value when the button is pressed. Note that the variable ISerRxValue is shown as the value of the last received byte, this is the parameter we wish to transmit. Now go back to the edit window and change the function as follows:

```
void TxLastRx()
{
    if (!RxFlag) return;           // Return if no byte received
    RxFlag=0;

    // void pSerialOut(unsigned char Tx);
    // - Transmit value to port
    pSerialOut(ISerRxValue);

    LED=0;           // Turn off LED
}
```

The complete Tutor_User.ASM file should now look like the listing below (Use the *File | Print* menu option to print the file). Check the file and edit any changes.

```
#include "C:\\Program Files\\FED\\WIZ-C\\Projects\\Tutor\\Tutor_Auto.h"
#include "Tutor.h"
//
// This file includes all user definable routines. It may be changed at
will as
```

```

// it will not be regenerated once the application has been generated for
the
// first time.
//

/*****
*****/
//
// Insert your interrupt handling code if required here.
// Note quick interrupts are used so code must be simple
// See the manual for details of quick interrupts.
//

void UserInterrupt()
{
    // Insert your code here

    #asmline SETPCLATH UserIntReturn,-1    ; SETPCLATH for interrupt routine
    #asmline goto UserIntReturn           ; Assembler - go back to interrupt
routine
}

/*****
*****/
//
// Insert your initialisation code if required here.
// Note that when this routine is called Interrupts will not be enabled -
the
// Application Designer will enable them before the main loop
//

void UserInitialise()
{
}

/*****
*****/
//
// Insert your main loop code if required here. This routine will be
called
// as part of the main loop code
//

void UserLoop()
{
}

//
// User occurrence code
//

//
// Occurrence - Switch Pressed
//

void TxLastRx()
{
    if (!RxFlag) return;                // Return if no byte received
    RxFlag=0;

    // void pSerialOut(unsigned char Tx);
    // - Transmit value to port
    pSerialOut(ISerRxValue);

    LED=0;
}

//
// Occurrence - Received a Byte
//

void LEDOn()
{
    RxFlag=1;                            // Show a byte has been received
    LED=1;                                // Turn on the LED
}

```

```
}

```

Now generate the application again by using the Ctrl and F9 keys or the button as described above. If you get errors then double click them in the Information window and correct the line with the error - use the listing above to see how they should read.

This is now the completed application which can be programmed into a PIC16F84 and run directly. However we can also simulate it and look at the results on simulated devices and the waveform analyser.

16.3 Simulation

It is not the intention of the introductory manual for WIZ-C to cover all the simulation capabilities of the environment which is covered in the later section. However we can check the operation of the program. It is possible to simulate with a stimulus file or with direct simulation of the external devices. We'll start with simulation of the external devices.

16.3.1 Switching screen layouts

There is a large amount of information provided on the screen and to aid users there are 3 main views :

Compact	Press ALT+C keys
Debugging	Press ALT+D keys
Editting	Press ALT+C keys

In compact mode all windows (Debug, Project, Editting and Information) are shown on screen. In Editing mode the debug window is hidden and most screen space is given to the edit window. In Debugging mode most space is given to the debug window.

FED recommend that users get used to using the ALT and C, D or E keys to rapidly switch views. Normally only the ALT+D and ALT+E modes will be use, the compact mode is provided for existing users and is similar to previous versions of our environments.


16.3.2 Simulating with external devices

Press ALT+D to switch to the debugging layout. You will already see a "device picture". This is a picture of the PIC with its pins shown. The pin colours show the state of ports. Red shows high and green low whilst the dark colours show the PIC is driving and light colours show the pins are operating as inputs.

WIZ-C has the capability to simulate LED's, switches, LCD displays and a number of other devices which might be connected to the PIC.

We'll start with the LED. Use the *Simulate / Add External Device* menu option. A dialog box will come up. In the External Device type box select LED. There are a number of parameters and values which may be selected for each device. For the LED most of these can be ignored apart from the connections. Under the Connections box there is a list box called Pins with two entries "Anode" and "Cathode". Select Anode and then use the Port box to select Port B, use the Bit box to select bit 5. Select Cathode and click the Connect Low option. This will connect the LED between Bit B5 and ground. Press the OK button. Note how the LED appears on the debugging window.


Next the push button. This time we'll use a short cut.

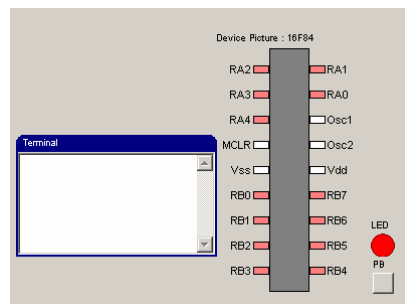
Click the Show Application Designer button at the top of the screen (). The Application designer has a button on it which will automatically generate an external device to match the

element. Click the Push Button element in the Element Store and then click the Create Device button which looks like this:



A push button will be created. Do the same for the serial element – click the element and then click the Create Device button.

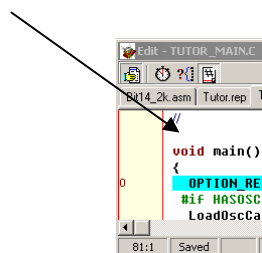
Hide the Application Designer by pressing the  button again. The new devices will be shown overlapped on the debugging window. It is possible to move them around by clicking on the title bar where the device name is shown and dragging the windows around. We adjusted them like this:



Note that the LED is illuminated. This is because the simulator assumes all unconnected inputs to be logic 1. Now run the simulation by using the *Simulate / Run* menu option (you could also press F9, or use the button of the running man on the toolbar).

The LED will go out. Click the terminal box (the cursor will appear) and type the letter A. Watch the LED go on, press the Push Button and it will go out transmitting the character back to the terminal. Experiment with other characters. Note that you may have to hold the push button down for a few (simulation) milli-seconds before it registers. Note how the RB4 input changes to pale green as the push button is pressed and how the Rb5 output changes from dark red to light red as the LED goes on and off.

Stop the program. Now look at the information bar – the yellow bar to the left of the edit window.



If the bar is not showing click the button to the top left of the edit window to turn it on (similarly click again to turn it off). Now the information bar can be set to show the address of each program line in the source, the time at which the line was last executed, or the number of times that it has been executed – click the buttons along the top of the edit window to select each option.

You can also determine how long a function or block of code takes to execute. Select a block of code – say the following block :

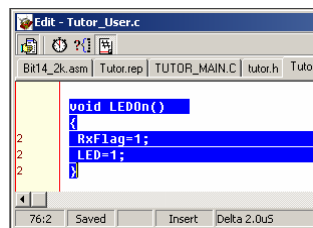
```
void LEDOn()
{
    RxFlag=1;                // Show a byte has been received
```

```

    LED=1;          // Turn on the LED
}

```

Now select the block of code with the mouse by clicking down before the void and dragging to the end of the function – you will see a box at the bottom of the edit window which will show the total time between the selected lines – in this case 2uS - the picture below shows the relevant part of the window:



You can drag over any area of code and the edit window will show the total time taken to execute that code provided that it has been simulated.

16.3.3 Simulating with a simulation file - using the waveform analyser

Next we will look at using a simulation file which will run alongside our external devices. Reset the processor by using the *Simulate / Reset Processor* menu option or press the reset button on the toolbar :



You may wish to switch back to the edit view using ALT+E. The first item is the simulation file - this will define the inputs to the program. Create a new blank file and save as type "Stimulus" with the file name "TutorInput.sti".

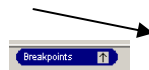
Select the project window and press Insert. A dialog box will open. Select Stimulus under the "Files of Type" box. Enter the filename "TutorInput" and press OK. Check that the file type is "Stimulus" in the Project File dialog box and press OK. Double click the file "TutorInput.STI" in the project window. We will generate a stimulus file which enters the character 41Hex at time 1mS, and then the key is pressed at time 5mS. To do this enter the following information (again the Simulator introductory manual covers this in more detail).

```

1m
serial9600-PORTE:0=41H      ; Enter character to Port B bit 0 at 1mS
5m
PORTE:4=0                   ; Press key at 5mS

```

Save the file by using the *File / Save All* menu option. Now we want to run the program until time 30mS and then stop to check the inputs and outputs. Press ALT+D to switch to debugging view. Bring up the breakpoint window by clicking the arrow on the minimised window:

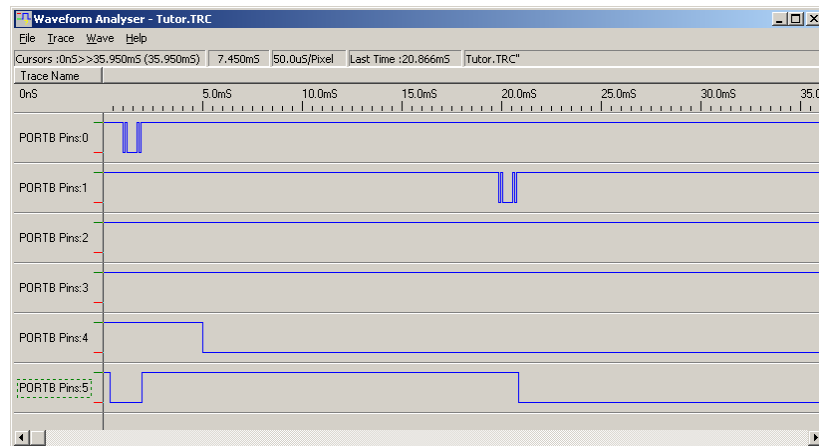


Right click the window and use the "Add Breakpoint" option. The Breakpoint Definition dialog box will be shown. Select "Break at Time", and then in the At Time box enter 30mS, click the OK button to define the breakpoint.

Now run the program by using the *Simulate / Run* menu option (you could also press F9, or use the button of the running man on the toolbar). Wait until the breakpoint is hit at 30mS. Use the *Tools / Examine Wave Window* menu option. The Wave Window "Define Trace Format" dialog box will be shown. In the Trace Name box select "Port B Pins", and then click the "Add as 8 line traces button". Resize the window to a comfortable size.

Finally press the F8 repeatedly button to zoom out and watch the received byte on Port B, bit 0, followed by Bit 5 going high (the LED turning on). Then at about 19mS the key press is detected (remember it is being debounced and takes a few milliseconds to detect it). You can see the received byte being transmitted on Port B, bit 1 and then the LED is turned off. The window can be copied to the clipboard or printed by using the options in the File menu for the Wave Window.

Here it is :



Run the simulation again - see how the use of the STI file operates with the external devices - the STI file has set the push button input low so until the button is pressed again the push button will automatically repeat every 250mS. Enter a character into the terminal and watch how it is repeated back automatically. Press the push button once to return to normal behaviour.

You are recommended to run through the simulator manual which follows this introductory manual.

16.4 Example 2

Digital clock operating to an LCD Display (in 10 lines of code)

In this example we shall show how to create a digital clock operating on an LCD display with less than 10 lines of code. Although in itself this project may not be that useful, it can operate in any project to provide subsidiary timing functions.

16.5 Digital Clock Element

The digital clock element operates a full digital clock based on timer 0 which holds counters for seconds, minutes, hours and day of the week from 0 to 6. Either a 12 or 24 hour clock may be maintained.

It maintains five variables - Secs, Mins, Hours, pm and Day. Each of these is updated at the correct time and an occurrence happens when each item changes enabling a 7 segment or LCD display to be updated.

There are 3 functions for setting the time. IncMin(); IncHour(); and IncDay(); IncMin increments the minute by one, triggers the Minute passing occurrence and resets the seconds

counters and internal counter chain to 0. It does not affect hours or days. IncHour() simply changes the hour without affecting seconds, minutes or days, but triggers the hour passing occurrence and IncDay() updates the day.

The accuracy of the clock is related to the overflow time of Timer 0. The faster that Timer 0 overflows the more accurate the clock. The internal counters are trimmed every minute and hour, and with a Timer 0 overflow of 1mS the accuracy is 1 part in 3.6×10^6 which is an order of magnitude more accurate than most crystals.

For this example we shall operate a 24 hour clock showing hours, minutes and seconds, days will not be displayed. There will be two push buttons to set the time - set minutes and set hours.

16.6 LCD Displays

16.6.1 Introduction

The LCD element provides functions are to drive an LCD module based on the Hitachi chip set. The functions handle the 4 bit interface, and the device timing to the module. They also read the module busy flag and hold future transfers whilst the module is still performing the last operation. Functions are provided to initialise the module, to transfer single characters to the module, to transfer LCD module commands, and to write strings to the module.

Such modules are the LM020, LM016, LM018 and LM032, however there are a number of other modules based on this chip which is numbered HD44780. The module is driven from any port, however the data bits (D4 to D7) must be connected to bits 4 to 7 of the same PIC port. We'll connect the display to ports D and E as follows:

LCD Module	LCD Port number	Pin Number (2 line display LM016L)
RS	D2	4
R/W	D3	5
E	E1	6
D4	D4	11
D5	D5	12
D6	D6	13
D7	D7	14
Vss	-	1
Vdd	-	2
Vo (LCD Supply)	-	3

16.6.2 Functions

The LCDSTRING function sends the supplied string to the display. Thus to write "HELLO" to the display then the following can be used:

```
LCDSTRING( "HELLO" )
```

A number of macros and functions are provided to drive the LCD Display. These are as follows:

```
void LCD(unsigned int Data);
```

Write Data to LCD as a character for display.

void LCDPrintAt(unsigned char x,unsigned char y);

Macro to print at line y, column x

e.g. `LCDPrintAt(5,0); // Print at line 0, column 5`

void LCDOnOff(unsigned char DisplayOn, unsigned char CursorOn, unsigned char Blink);

Macro to set up display. DisplayOn is 1 to enable the display, or 0 to turn it off, CursorOn is 1 to enable the display, or 0 to turn it off, Blink is 1 if the Cursor is to blink

e.g. `LCDOnOff(1,1,1); // Cursor on blinking`

void LCDShift(unsigned char Cursor, unsigned char Right);

Macro to shift display Left or Right, Cursor is 1 or 0 to control shift with the cursor. Right is 1 for a right shift or 0 for a left shift.

e.g. `LCDShift(1,0); // Shift display left 1 character`

void LCDClear();

Clear the display, return print position to 0,0.

16.7 Digital Clock Application

The application requires the digital clock element, the LCD element, and two push button elements. It will operate on the 16F877 - but in fact will run on most PIC's. Use the *File / Open/New Project* menu option to create a new project - or open this example which is included in the standard WIZ-C installation in the projects directory. Call the project DigClock.

Connect the elements as follows:

16.7.1.1 LCD

The LCD element is on the displays tab. It looks like this :



Once the LCD element is selected by double clicking it, it can be set up by connecting the pins to the pins of the F877. The only parameter for the LCD is the number of lines which needs setting to match the display. It does not need initialising.

16.7.1.2 Digital Clock

The digital clock element is on the Timers tab. It looks like this:



The only parameter is 12 or 24 hour selection. For this clock we'll use 24Hour. Now click on the Occurrences tab. Every time that the time changes (seconds, hours or minutes) then we'll update the display. So add a function call for SecPass, MinPass and HourPass. We'll use the same function for each - called UpDate. Ignore the Days for the moment.

16.7.1.3 Switches

Add two pushbutton elements for setting the hours and minutes. Connect one to RB0 and one to RB1. Now when a push is detected the minute (or hour) will be incremented. Right click on the digital clock element on the Timers tab. Select the "Help on Selected Element". This will bring up the help file. Look at the Public Calls and Variables section. Note that there is a function to increment minutes (which also clears the seconds counter) called IncMin, and one to increment the Hour. Return to the Application Designer. Add IncMin to the list of occurrences for RB0, and add IncHour to the list of occurrences for RB1. Now we need to write no code directly for the switch presses - the switch presses will translate directly to increment the minute and hour counters. On incrementing them the Digital Clock element will generate occurrences to update the display.

16.7.1.4 User Code

Generate the application (use Ctrl+F9).

Open the DigClock_User.C file. For the clock we can set up a welcome message on power up by including code in the UserInitialise() function as follows:

```
void UserInitialise()
{
    OPTION_REG&=0x7f;          // Port B pull ups
    // void LCDString(char *str);
    // - Write a string to the LCD
    LCDString("Digital Clock"); // Welcome message
    Wait(2000);                 // Wait for 2 seconds
    LCDClear();                 // Clear the display
}
```

Now to print the time to the display we need a function to convert an 8 bit number to a string. Use the *Help | Compiler Contents* menu option to open the C Compiler help file. Click Library Reference and then String Print Functions to bring up help on the functions which print numbers to strings. CPrintString is the function that we can use, however it does not print a leading zero if the number is less than 10. Add the following function to the bottom of the DigClock_User.c file which will print a number with a leading zero if necessary:

```
//
// Print a 2 digit number with a leading 0
//
void RJPrint(unsigned char v,char *s)
{
    if (v<10) { *s='0'; s++; }
    cPrtString(s,v);
}
```

Finally we need to write the UpDate() function which will print the time to the display. Add this function to the bottom of the file:

```
//
// Print the time on the display
//
void UpDate()
{
    char Ds[12];          // String to display

    RJPrint(Hours,Ds); Ds[2]=': ';          // Hours
    RJPrint(Mins,Ds+3); Ds[5]=': ';        // Minutes
    RJPrint(Secs,Ds+6);                    // Seconds
    LCDPrintAt(0,0);
    LCDString(Ds);                        // Print time to LCD
}
```

This is quite straightforward, when the display is to be updated it prints the time to the first row, first column.

16.7.1.5 Final application

Generate the application (use Ctrl+F9). The project may be simulated using the external devices. Select the LCD on the application designer and press the light bulb to generate the LCD device. Set the display to 2 lines by 16 rows (this is owing to an anomaly with the method of operation of LCD displays with one row). Add push buttons for hours and minutes connected to RB1 and RB0 respectively. Run the program with update rate set to 20000. Watch the display and press the hours and minutes buttons. Note that the simulation is far behind real time owing to the use of a 20MHz clock. You can simulate faster than real time by using the application designer and setting the oscillator rate to 1MHz (actual simulation speed will depend on the speed of the PC - this manual was written around a simulation running on a 266MHz Pentium).

If the application is to be simulated without external devices then the input D7 for the display should be set to 0 which will make the LCD library routines believe that a display is acknowledging. For our example this could be achieved with the following line in a STI file:

```
PORTD:7=0 ; Set port D bit 7 to zero
```

The final application file DigClock_User.c looks like this:

```
#include "C:\\Program Files\\FED\\WIZ-
C\\Projects\\DigClock\\DigClock_Auto.h"
#include <Delays.h>
#include <Strings.h>

//
// This file includes all user definable routines. It may be changed at
// will as
// it will not be regenerated once the application has been generated for
// the
// first time.
//

//
// *****
// *****
//
// Insert your interrupt handling code if required here.
// Note quick interrupts are used so code must be simple
// See the manual for details of quick interrupts.
//

void UserInterrupt()
{
    // Insert your code here

    #asmline goto UserIntReturn      ; PIC Assembler - go back to interrupt
routine
}

//
// *****
// *****
//
// Insert your initialisation code if required here.
// Note that when this routine is called Interrupts will not be enabled -
// the
// Application Designer will enable them before the main loop
//

void UserInitialise()
{
    OPTION_REG&=0x7f;           // Port B pull ups
    // void LCDString(char *str);
    // - Write a string to the LCD
    LCDString("Digital Clock");
    Wait(2000);
    LCDClear();
}
```

```
//
*****
****
//
// Insert your main loop code if required here. This routine will be
called
// as part of the main loop code
//

void UserLoop()
{
}

//
// User occurrence code
//

//
// Print a 2 digit number with a leading 0
//
void RJPrint(unsigned char v,char *s)
{
    if (v<10) { *s='0'; s++;}
    cPrtString(s,v);
}

//
// Print the time on the display
//
void UpDate()
{
    char Ds[12];          // String to display

    RJPrint(Hours,Ds); Ds[2]=': ';          // Hours
    RJPrint(Mins,Ds+3); Ds[5]=': '; // Minutes
    RJPrint(Secs,Ds+6);          // Seconds
    LCDPrintAt(0,0);
    LCDString(Ds);              // Print time to LCD
}
```

16.7.1.6 Including the day

Adding the day is very straightforward. A new push button element is required - connect to RB2 and couple the occurrence to IncDay to set the day. Add UpDate to the occurrence for DayPass for the Digital Clock element. Change the UpDate function as follows:

```
//
// Print the time on the display
//
const char *DayStr[]=
{
    "Sun ",
    "Mon ",
    "Tue ",
    "Wed ",
    "Thu ",
    "Fri ",
    "Sat "
};

void UpDate()
{
    char Ds[12];          // String to display

    RJPrint(Hours,Ds); Ds[2]=': ';          // Hours
    RJPrint(Mins,Ds+3); Ds[5]=': '; // Minutes
    RJPrint(Secs,Ds+6);          // Seconds
    LCDPrintAt(0,0);
    LCDString(DayStr[Day]);          // Print day to LCD
    LCDString(Ds);              // Print time to LCD
}
```

The DayStr array is set up in ROM to minimise RAM usage, the correct item from the array is printed before the rest of the time string.

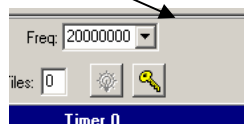
16.7.1.7 Taking it further

Try changing the clock to operate on a 12 hour clock with a pm indicator.
Make it into an alarm clock.
Extend into a multi-function timer.

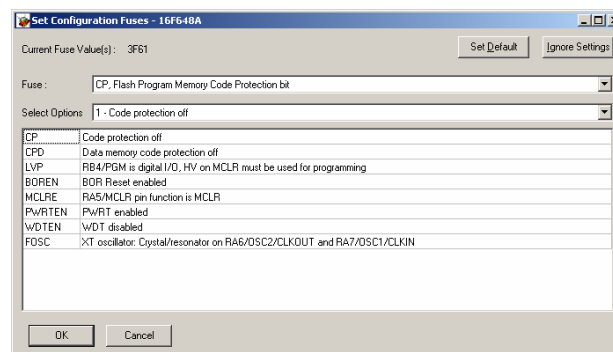
16.8 Setting the configuration fuses :

The C Compiler provides support for simple set up of the configuration fuses which is essential to do before running the program.

To bring up the Configuration fuses dialog use the Project | Set configuration fuses menu option, or in WIZ-C use the button on the application designer :



This will bring up the Set Configuration fuses dialog box :



This is very straightforward to use, and is described in section 3.10 of the C Compiler manual.

17 Appendix A - Real time programming source

17.1 File : Timer.h

```

// #define DEBUGGING

#ifdef MAIN
#define EXTERN
#else
#define EXTERN extern
#endif

#ifdef DEBUGGING
const int SECONDS=1;           // Number of overflows per second
const int MINUTES=60;          // Number of overflows per second
#else
const int SECONDS=610;         // Number of overflows per second
const int MINUTES=36621;       // Number of overflows per second
#endif

const int INITKDELAY=SECONDS/20/2; // Delay of 1/2S before 1st key
repeat
const int REPKDELAY=SECONDS/20/5; // Delay of 1/5S between key repeats
const int NOACTMENUTO=30;         // Time before menu times out

EXTERN unsigned char Second;      // Current time
EXTERN unsigned char Minute;
EXTERN unsigned char Hour;
EXTERN unsigned char Day;
EXTERN unsigned char Menu;
EXTERN unsigned char Item;
EXTERN unsigned char MenuTimeOut;

// Structures & enumerations

enum Flags {
    T0Overflow=1,           // Set when timer 0 has overflowed
    SecPassed=2,            // Set when one second has passed
    MinPassed=4,            // Set when one second has passed
    KeyPressed=8            // A button has been pressed
};

EXTERN unsigned char StateFlag;

enum Keys {
    NO_KEY=0xff,            // No button is pressed
    SET_KEY,               // Enter button is pressed
    ITEM_KEY,              // Next button is pressed
    UP_KEY                 // Up button is pressed
};

EXTERN unsigned char Key;      // Last key pressed
EXTERN unsigned int KeyDelay;  // Delay before key repeat

struct TimerEvent {
    unsigned char Day;         // Day of event (has value 8 for every
day)
    unsigned char Hour;       // Hour of event
    unsigned char Minute;     // Minute of event
    unsigned char BitAction;   // Action and bit to set
};

EXTERN TimerEvent Timers[16]; // Array of timer events

// Functions

void AddTime(unsigned char Fast); // Increment the time
void CheckEvent();               // Check if timer event is reached
void Display();                  // Update the display
void DisplayInit();              // Initialise display
void DoKeyPress();               // Action a key press

```

```
void DoT0Overflow();           // Action whenever Timer 0
overflows
void LCDClear();              // Clear LCD display
void ReadBut();              // Read button
void TimerInit();            // Initialise the system
char *TimeString(char *ws,unsigned char Hour,unsigned char Min,
                unsigned char Sec,unsigned char Day,unsigned char
ShowSec);

// Variables

EXTERN unsigned int SecCount;    // Counts overflows for seconds
EXTERN unsigned int MinCount;    // Counts overflows for minutes
```


17.2 File : Timer.c

```

#include <P16F877.h>
#define MAIN
#include "Timer.h"
#_config _CP_OFF & _BODEN_ON & _WDT_OFF & _PWRTE_ON & _HS_OSC & _LVP_OFF

void main()
{
    TimerInit();           // Initialise the system
    DisplayInit();

    while(1)               // Loop forever
    {
        if (StateFlag&T0Overflow)
            {DoT0Overflow(); StateFlag&=~T0Overflow;}

        if (StateFlag&MinPassed)
            {CheckEvent(); StateFlag&=~MinPassed;}

        if (StateFlag&KeyPressed)
            {DoKeyPress(); StateFlag&=~KeyPressed;}
    }

    //
    // Interrupt handler
    //

    void Interrupt()
    {
        if (INTCON&(1<<T0IF))                // Test Timer 0
            overflow                          // overflow
            {INTCON&=~(1<<T0IF); StateFlag|=T0Overflow;}
    }

    //
    // Initialise the system
    //
    void TimerInit()
    {
        StateFlag=0;                          // Reset flags
        PORTB=0;                               // Port B outputs all off
        TRISB=0;                               // Drive all outputs of TRISB low
        TRISD&=0xfe;                          // Bit 0 of PORTD to write (for
        keypresses)
        PORTD&=0xfe;                          // Bit 0 of PORTD to value 0
        OPTION_REG=4;                         // Timer 0 internal, divide by 16
        TMR0=0;                               // Clear timer 0
        Key=NO_KEY;                           // No key pressed
        Day=Hour=Minute=Second=0;             // Set time to midnight, Sunday
        SecCount=MinCount=0;                  // Clear counters
        INTCON=(1<<GIE)|(1<<T0IE);            // Enable timer 0 interrupts
    }

    //
    // Actions when timer 0 overflows
    //
    void DoT0Overflow()
    {
        SecCount++;
        MinCount++;
        if (!(SecCount&0x1f)) ReadBut(); // Read button every 50ms
        if (SecCount==SECONDS) {SecCount=0; AddTime(0); Display();}
        if (MinCount==MINUTES) SecCount=MinCount=0;
    }

    //
    // Time has passed.
    // The Fast value is set to 1 if minutes are to be increased
    // The Fast value is set to 2 if hours are to be increased
    // The Fast value is set to 3 if days are to be increased

```

```

// directly - this is used for setting the time.
//
void AddTime()
{
    if (MenuTimeOut && !--MenuTimeOut) Menu=0xff; // Return to standard
display

    Second++;
    if (Second!=60) return;
    Second=0;
    Minute++;
    StateFlag|=MinPassed;
    if (Minute!=60) return;
    Minute=0;
    Hour++;
    if (Hour!=24) return;
    Hour=0;
    Day++;
    if (Day==7) Day=0;
}

//
// Check if an event time has been reached
//

void CheckEvent()
{
    unsigned char i;
    TimerEvent *tep; // Pointer to timer event

    tep=Timers;

    for(i=16; i; i--)// Loop to check each timer
    {
        if (tep->Minute==Minute && tep->Hour==Hour && (tep->Day==7 || tep-
>Day==Day))
        {
            unsigned char Mask; // Holds mask for bit

            Mask=1<<(tep->BitAction&7);
            if (tep->BitAction&8) PORTB|=Mask; // Turn on bit
            else PORTB&=~Mask; // Turn off bit
        }
        tep++; // Next timer event
    }
}

```

17.3 File : TimerHMI.c

```

#include <pic.h>
#include <Displays.h>
#include <Delays.h>
#include <Strings.h>
#include "Timer.h"

const int LCDPORT=&PORTD; // Connect module to port D
const int LCDEPORT=&PORTE;
const int LCDEBIT=1;
const int LCDRSPORT=&PORTD;
const int LCDRSBIT=2;
const int LCDRWPORT=&PORTD;
const int LCDRWBIT=3;

const char *Days[] = // Array of string representing days
{
    "Su",
    "Mo",
    "Tu",
    "We",
    "Th",
    "Fr",
    "Sa",
    "Ev"
};

void DisplayInit()
{
    Menu=0xff; // Initial value of menu
    MenuTimeOut=0; // Delay before returning menu to std
    display
    Item=0; // Item - hours, mins, day, bit, event
    LCD(-2); // Initialise display to 2 lines
    LCDClear(); // clear display
    LCDString("Timer"); // Start up string
    #ifndef DEBUGGING
        Wait(1000); // Delay 1 s before starting
    #endif
}

//
// Display values on screen
//
void Display()
{
    char ws[17]; // String to hold display values
    unsigned char i;

    LCDClear(); // Clear display

    if (Menu==0xff) // Normal display (time & State of outputs)
    {
        LCDPrintAt(0,0); // 1st Line
        TimeString(ws,Hour,Minute,Second,Day,1); // Show the time
        LCDString(ws);
        LCDPrintAt(0,1); // 2nd line
        for(i=0; i<8; i++) // Show state of all
            outputs
            {LCD(((PORTB>>i)&1)+'0'); LCD(' ');} // Display 1 or 0 for
        output
        return;
    }

    if (Menu<=15) // Setting an event
    {
        LCDPrintAt(0,0); LCDString("Set Ev "); // Display "Set Ev"
        cPrtString(ws,Menu+1); LCDString(ws); // Display event number

        LCDPrintAt(0,1);
        TimeString(ws,Timers[Menu].Hour,Timers[Menu].Minute,0,
            Timers[Menu].Day,0); // Show the time
    }
}

```

```

    LCDString(ws); LCD(' ');

    LCD(((Timers[Menu].BitAction)&7)+'0'); LCD(' ');    // Display event
number

    if (Timers[Menu].BitAction&8) LCD('1');            // Action is to set
bit
    else LCD('0');

    LCDPrintAt(10,0);                                // Print position for item
}

if (Menu==16)    // Set the time
{
    LCDPrintAt(0,0); LCDString("Set Time ");    // Display "Set Time"

    LCDPrintAt(0,1);
    TimeString(ws,Hour,Minute,0,Day,1);    // Show the time

    LCDString(ws);
    LCDPrintAt(9,0);    // Print position for item
}

    // Now print the item that we are setting
switch(Item)
{
    case 0: i='D'; break;
    case 1: i='H'; break;
    case 2: i='M'; break;
    case 3: i='B'; break;
}
LCD(i);    // Print the item
}

//
// Print a time to a string
//

char *TimeString(char *ws,unsigned char Hour,unsigned char Min,
                unsigned char Sec,unsigned char Day,unsigned char ShowSec)
{
    unsigned char at=0;

    strcpy(ws,Days[Day]);    // Print day to string
    ws[2]=' '; at=3;    // Space (gap)
    if (Hour<10) ws[at++]=' ';    // Hour leading space
    cPrtString(ws+at,Hour);    // Hours
    ws[5]=': '; at=6;    // Colon
    if (Min<10) ws[at++]='0';    // Minute leading zero
    cPrtString(ws+at,Min);    // Minute
    ws[8]=0;    // End of string
    if (ShowSec)    // Display seconds ?
    {
        ws[8]=': '; at=9;
        if (Sec<10) ws[at++]='0';    // Minute leading zero
        cPrtString(ws+at,Sec);
    }
    ws[11]=0;    // End of string marker
    return ws;
}

//
// Read a button on a timer interrupt
//

void ReadBut()
{
    unsigned char NewKey=NO_KEY;

    TRISD|=0xf0;    // Upper 4 bits of PORT D to read
    if (!(PORTD&0x80)) NewKey=SET_KEY;    // Set key
    if (!(PORTD&0x40)) NewKey=ITEM_KEY;    // Item key
    if (!(PORTD&0x20)) NewKey=UP_KEY;    // Up key

    if (NewKey==NO_KEY) {Key=NO_KEY; return;}    // No key press

    if (NewKey!=Key)

```

```

    {
        KeyDelay=INITKDELAY;
        StateFlag|=KeyPressed;
        Key=NewKey;
        return;
    }

    if (!KeyDelay)
    {
        KeyDelay=REPKDELAY;
        StateFlag|=KeyPressed;
        return;
    }

    KeyDelay--;
}

//
// Handle a key press
//

void DoKeyPress()
{
    if (Key==SET_KEY)
    {
        Menu++;
        if (Menu==17) Menu=0xff;
        Item=0;
    }

    if (Key==ITEM_KEY)
    {
        Item++;
        if (Menu<=15)
        {if (Item==4) Item=0;}
        else
        {if (Item==3) Item=0;}
    }

    if (Key==UP_KEY)
    {
        if (Menu<=15)
        switch(Item)
        {
            case 0 : Timers[Menu].Day++;
                     if (Timers[Menu].Day==8) Timers[Menu].Day=0;
                     break;
            case 1 : Timers[Menu].Hour++;
                     if (Timers[Menu].Hour==24) Timers[Menu].Hour=0;
                     break;
            case 2 : Timers[Menu].Minute++;
                     if (Timers[Menu].Minute==60) Timers[Menu].Minute=0;
                     break;
            case 3 : Timers[Menu].BitAction++;
                     break;
        }
    }
    if (Menu==16)
    switch(Item)
    {
        case 0 : Day++; if (Day==7) Day=0; break;
        case 1 : Hour++; if (Hour==24) Hour=0; break;
        case 2 : Minute++; if (Minute==60) Minute=0;
                     Second=0;
                     break;
    }
}
Display();
MenuTimeOut=NOACTMENUTO;
}

```

18 Appendix B - List of FED PIC keywords

...	One or more parameter may be supplied to a function
auto	Not applicable to FED PIC C
break	Cuts out of innermost loop
case	Case statement within a switch
catch	Not applicable to FED PIC C
char	8 bit signed integer
const	Defines a constant or non changeable parameter
continue	Continue innermost loop
default	Default case statement
do	Loop
double	As float
else	Alternate to if statement
enum	Starts list of enumerated variables
extern	Defines but does not allocate variable space
float	32 bit floating point number
for	Starts loop
goto	Switch control
if	Conditional statement
int	16 bit signed integer within FED PIC C
interrupt	Keyword used for 3 rd party compatibility.
long	32 bit signed integer within FED PIC C
mutable	Not applicable to FED PIC C
pointed	Specifies that a function may be called through a pointer
ram	FED PIC C extension to define 8 bit pointer for 12 or 16 series devices, or to use FSR accumulator for the 18 series devices.
register	Directs the compiler to place items in the bottom page if possible
return	return from function
rom	FED PIC C extension to define 8 bit pointer (default pointer length)
short	16 bit signed integer within FED PIC C
signed	Define value/variable as signed
static	Forces local variable into global space
struct	Starts a structure definition
switch	Begins a switch statement
throw	Not applicable to FED PIC C
try	Not applicable to FED PIC C
typedef	Defines a type
union	Starts a union definition
unsigned	Define value/variable as unsigned
void	Type unknown, zero length or not applicable
volatile	Define register as independently changing
while	Begins loop

19 Appendix C - List of WIZ-C Professional standard library functions

Function	Area
AddTx	Interrupt Driven Serial Port
Checksum	String Functions
cos	Maths Routines
cPrintToString	String Print Functions
e	Maths Routines
exp	Maths Routines
exponent	Maths Routines
fabs	Maths Routines
fprintf	printf Functions
fprintfsm	printf Functions
fPrtString	Maths Routines
GetRxSize	Interrupt Driven Serial Port
GetTxSize	Interrupt Driven Serial Port
IIRead	I2C Routines
IISWrite	I2C Routines
iPrintToString	String Print Functions
IRRx	IRDA IR Routines
IRRxVal	IRDA IR Routines
IRTx	IRDA IR Routines
KeyScan	Hex Keypad
LCDc	LCD
LCD	LCD
LCDString	LCDString
LN2	Maths Routines
LN10	Maths Routines
log	Maths Routines
log10	Maths Routines
LOG2_10	Maths Routines
lPrintToString	String Print Functions
memcpy	String Functions
pClockDataIn	ClockDataIn
pClockDataOut	ClockDataOut
PI	Maths Routines
pow	Maths Routines
pow10	Maths Routines
PowerSeries	Maths Routines
printf	printf Functions
pSerialIn	SerialIn
pSerialOut	SerialOut
QuickStop	I2C Routines
rand	Random Numbers
RC5Rx	RC5 IR Routines
RC5Transmit	RC5 IR Routines
rChecksum	String Functions
rcPrintToString	String Print Functions
ReadEEData	EEPROM Routines
riPrintToString	String Print Functions
rlPrintToString	String Print Functions
rmemcpy	String Functions
rstrcat	String Functions
rstrchr	String Functions

Function	Area
rstrcmp	String Functions
rstrcpy	String Functions
rstrlen	String Functions
rstrlwr	String Functions
rstrupr	String Functions
rvstrcpy	String Functions
SerIntHandler	Interrupt Driven Serial Port
SerIntInit	Interrupt Driven Serial Port
sin	Maths Routines
sprintf	printf Functions
sprintfsm	printf Functions
srand	Random Numbers
sqrt	Maths Routines
strcat	String Functions
strchr	String Functions
strcmp	String Functions
strcpy	String Functions
strlen	String Functions
strlwr	String Functions
tan	Maths Routines
vstrcpy	String Functions
Wait	Wait
WaitRx	Interrupt Driven Serial Port
WriteEEData	EEPROM Routines

20 Appendix D - List of supported PIC's at December 2007

12C671	16C715	16F690	16F916	18F2610	18F6527
12C672	16C71A	16F737	16F917	18F2620	18F65J10
12CE673	16C72	16F747	18C242	18F2680	18F65J15
12CE674	16C73	16F767	18C252	18F4220	18F65J50
12F629	16C73A	16F777	18C442	18F4320	18F6620
12F675	16C73B	16F785	18C452	18F4321	18F6622
12F683	16C74	16F818	18F010	18F4331	18F6627
14000	16C74A	16F819	18F020	18F4410	18F66J10
16C554	16C76	16F820	18F1220	18F442	18F66J15
16C556	16C765	16F821	18F1320	18F4420	18F66J50
16C558	16C77	16F822	18F2220	18F4423	18F66J55
16C61	16C773	16F83	18F2320	18F4431	18F6720
16C62	16C774	16F84	18F2331	18F4455	18F6722
16C620	16C84	16F87	18F2410	18F4458	18F67J10
16C621	16C923	16F870	18F242	18F448	18F67J50
16C622	16C924	16F871	18F2420	18F4480	18F8390
16C62A	16ce623	16F872	18F2423	18F4510	18F8490
16C63	16ce624	16F873	18F2431	18F4515	18F8520
16C63A	16ce625	16F873A	18F2455	18F452	18F8527
16C64	16CR83	16F874	18F2458	18F4520	18F85J10
16C641	16CR84	16F874A	18F248	18F4523	18F85J15
16C642	16F627	16F876	18F2480	18F4525	18F85J50
16C64A	16F628	16F876A	18F2510	18F4550	18F8620
16C65	16F630	16F877	18F2515	18F4553	18F8622
16C65A	16F631	16F877A	18F252	18F458	18F8627
16C66	16F648A	16F88	18F2520	18F4580	18F86J10
16C661	16F676	16F882	18F2523	18F4585	18F86J15
16C662	16F677	16F883	18F2525	18F4610	18F86J50
16C67	16F684	16F884	18F2550	18F4620	18F86J55
16C70	16F685	16F886	18F2553	18F4680	18F8720
16C71	16F687	16F887	18F258	18F6390	18F8722
16C710	16F688	16F913	18F2580	18F6490	18F87J10
16C711	16F689	16F914	18F2585	18F6520	18F87J50

21 Appendix E – Configuration Fuses

Configuration fuses for the 14 bit core devices
 =====

Probably one of the biggest causes of programs failing to run on real hardware even though they are fine on the simulator is fuse settings. We'll take a look at both 14 bit and 16 bit fuse settings, starting here with the 14 bit core devices, that is those starting 12F, 12C, 16F and 16C.

The configuration fuses set those parameters of the device which (generally) should not or must not be set by the running program, for example the oscillator settings which if set wrongly will not even allow the program to run.

The configuration bits are normally documented in the device data sheet in the section "Special features of the CPU". For 14 bit core devices the word is in memory at location 2007hex, although some devices have 2 fuses, we'll take a look at those in a later FAQ.

For an example we'll look at the bits in the 16F819 device, and then see how we can include this information in the C source file within the compiler or WIZ-C. We'll look at typical values for this device. This is the definition of the bits (remember the default value is usually 1):

```

bit 13 CP: Flash Program Memory Code Protection bit
          1 = Code protection off
          0 = All memory locations code protected
          >>> Typically leave this bit set to 1.

bit 12 CCPMX: CCP1 Pin Selection bit
          1 = CCP1 function on RB2
          0 = CCP1 function on RB3
          >>> Unless we want to change this for some reason the default value (1) is
best.

bit 11 DEBUG: In-Circuit Debugger Mode bit
          1 = In-Circuit Debugger disabled, RB6 and RB7 are general purpose I/O
pins
          0 = In-Circuit Debugger enabled, RB6 and RB7 are dedicated to the
debugger
          >>> Set this to 1 unless using the Microchip ICD

bit 10-9 WRT1:WRT0: FLASH Program Memory Write Enable bits
For PIC16F819:
          11 = Write protection off
          10 = 0000h to 01FFh write protected, 0200h to 07FFh modified by EECON
control
          01 = 0000h to 03FFh write protected, 0400h to 07FFh modified by EECON
control
          00 = 0000h to 05FFh write protected, 0600h to 07FFh modified by EECON
control
          >>> Leave set to 11 unless required.

bit 8 CPD: Data EE Memory Code Protection bit
          1 = Code protection off
          0 = Data EE memory locations code protected
          >>> Leave set to 1 unless required

bit 7 LVP: Low Voltage Programming Enable bit
          1 = RB3/PGM pin has PGM function, low voltage programming enabled
          0 = RB3/PGM pin has digital I/O function, HV on MCLR used for
programming
          >>> Normally clear this bit to allow RB3 to be used as a digital I/O. Very few
programmers only support LVP.

bit 6 BOREN: Brown-out Reset Enable bit
          1 = BOR enabled
          0 = BOR disabled
          >>> Leave this bit set, BOR is a useful function.

bit 5 MCLRE: RA5/MCLR Pin Function Select bit
          1 = RA5/MCLR pin function is MCLR
          0 = RA5/MCLR pin function is digital I/O, MCLR internally tied to VDD
          >>> We usually leave this bit set unless we really need MCLR.

```

```

bit 3 PWRTEN: Power-up Timer Enable bit
        1 = PWRT disabled
        0 = PWRT enabled
    >>> Normally clear this bit, it makes the device more robust

bit 2 WDTEN: Watchdog Timer Enable bit
        1 = WDT enabled
        0 = WDT disabled
    >>> Clear this unless you are using the Watchdog - COMMON SOURCE OF ERRORS

bit 4, 1-0 FOSC2:FOSC0: Oscillator Selection bits
        111 = EXTRC oscillator; CLKO on RA6/OSC2/CLKO pin
        110 = EXTRC oscillator; port I/O on RA6/OSC2/CLKO pin
        101 = INTRC oscillator; CLKO on RA6/OSC2/CLKO & port I/O on
RA7/OSC1/CLKI
        100 = INTRC oscillator; port I/O function on RA6/OSC2/CLKO &
RA7/OSC1/CLKI
        011 = EXTCLK; port I/O function on RA6/OSC2/CLKO pin
        010 = HS oscillator
        001 = XT oscillator
        000 = LP oscillator

>>> Setting the oscillator bit is vital. To give some indication we find most users
have one of 3 configurations.

    4MHz external crystal or resonator - use XT oscillator. (Value 001)
    20MHZ external crystal or resonator - use HS oscillator. (Value 010)
    Internal RC oscillator (4MHz) - use INTRC with port I/O function (Value 100)

```

So now we have the bit values for the fuses. For a 4MHz crystal oscillator these are

```

CP|CCPMX|DEBUG|WRT1|WRT0|CPD|LVP|BOREN|MCLR|FOSC2|PWRTEN|WDTEN|FOSC1|FOSC0
1 1 | 1 1 1 1 | 0 1 1 0 | 0 0 0 1

```

In hex this is 0x3f61.

So in our programmer we can select this fuse value.

Alternatively it can be set up in the C Compiler. For a 14 bit core device the easiest way to do this is simply use the `__config` pragma directive with the hex value:

```
#pragma __config 0x3f61
```

This can appear anywhere in any one of the source files, WIZ-C users may like to standardise on putting the value at the top of the `xxx_user.c` file.

You can also use symbolic constants. It is best to open the header file for the device (P16F819.h) and look at the values to understand how to use them for each device. For the 16F819 the header file for configuration fuses looks like this:

```

// Configuration Bits

const int _CP_ALL=0x1FFF;
const int _CP_OFF=0x3FFF;
const int _CCP1_RB2=0x3FFF;
const int _CCP1_RB3=0x2FFF;
const int _DEBUG_OFF=0x3FFF;
const int _DEBUG_ON=0x37FF;
const int _WRT_ENABLE_OFF=0x3FFF;
const int _WRT_ENABLE_512=0x3DFF;
const int _WRT_ENABLE_1024=0x3BFF;
const int _WRT_ENABLE_1536=0x39FF;
const int _CPD_ON=0x3EFF;
const int _CPD_OFF=0x3FFF;
const int _LVP_ON=0x3FFF;
const int _LVP_OFF=0x3F7F;
const int _BODEN_ON=0x3FFF;
const int _BODEN_OFF=0x3FBF;
const int _MCLR_ON=0x3FFF;
const int _MCLR_OFF=0x3FDF;
const int _PWRTEN_OFF=0x3FFF;
const int _PWRTEN_ON=0x3FF7;
const int _WDT_ON=0x3FFF;
const int _WDT_OFF=0x3FFB;
const int _EXTRC_CLKOUT=0x3FFF;

```

```
const int _EXTRC_IO=0x3FFE;
const int _INTRC_CLKOUT=0x3FFD;
const int _INTRC_IO=0x3FFC;
const int _EXTCLK=0x3FEF;
const int _HS_OSC=0x3FEE;
const int _XT_OSC=0x3FED;
const int _LP_OSC=0x3FEC;
```

The way to use these values is to AND the bits together using the C operator &. Our example looks like this:

```
#pragma __config _CP_OFF & _CCP1_RB2 & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF&
\
    _LVP_OFF & _BODEN_ON & _WDT_OFF & _MCLR_ON & _PWRTE_ON &
_XT_OSC
```

Note the '\\' at the end of the first line which joins the lines together. This is horrible ! In fact we don't need to include default values at all, so we could use :

```
#pragma __config _LVP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC
```

This is much better !

One final point - please note the double underscore (__) before the word config. This is for historic reasons, it will not create an error if spelt wrongly as the compiler (correctly) ignores #pragma statements that it does not recognise.

Configuration fuses for the 16 bit core devices
=====

In FAQ 11 we looked at configuration fuses for the 14 bit core devices, in this FAQ we'll take a look at 16 bit core devices.

18 series devices have up to 7 words or 14 bytes of configuration memory. These are referred to symbolically as _CONFIG1L, _CONFIG1H up to _CONFIG7H, later devices may have further configuration fuses.

Unfortunately not many PIC programmers allow the user to select device fuses by check boxes. Equally it is quite hard to work out the hex patterns for up to 14 configuration bytes. Therefore for these devices it is essential to consult the header file as well as the data sheet. Looking at the header file for the 18F452 for Configuration byte 2H :

```
//Configuration Byte 2H Options

#define _WDT_ON_2H 0xFF          // Watch Dog Timer enable
#define _WDT_OFF_2H 0xFE
#define _WDTPS_128_2H 0xFF      // Watch Dog Timer PostScaler count
#define _WDTPS_64_2H 0xFD
#define _WDTPS_32_2H 0xFB
#define _WDTPS_16_2H 0xF9
#define _WDTPS_8_2H 0xF7
#define _WDTPS_4_2H 0xF5
#define _WDTPS_2_2H 0xF3
#define _WDTPS_1_2H 0xF1
```

Note that, as for the 14 bit core devices we must AND these values together. Microchip have been helpful in appending the register number to each constant for us as well - note the _2H at the end of each name. So to set the watchdog to OFF, but also the postscalar to 128 (in case we turn it on at some later date) use the following construct :

```
#pragma __config _CONFIG2H,_WDT_OFF_2H & _WDTPS_128_2H
```

alternatively the non-ANSI, but easier version is :

```
#__config _CONFIG2H,_WDT_OFF_2H & _WDTPS_128_2H
```

again this can be placed in any source file, WIZ-C users can use the top of the xxx_user.c file as a standard. Note how the address of the configuration register must be provided when there is more than one register.

So to look at a typical 18F452 oscillator configuration for a device running at 20MHz (i.e. an HS oscillator) with no Watchdog, and no Low Voltage Programming pin, with the Power up timer and Brown Out Reset timers enabled :

```
#__config _CONFIG1H,_HS_OSC_1H
```

```
#__config _CONFIG2L,_PWRT_ON_2L&_BOR_ON_2L  
#__config _CONFIG2H,_WDT_OFF_2H  
#__config _CONFIG4L,_LVP_OFF_4L
```

Compare this to the data sheet to see how we defined these values.

Note that we don't need to define anything for registers which are to be set to their default values.